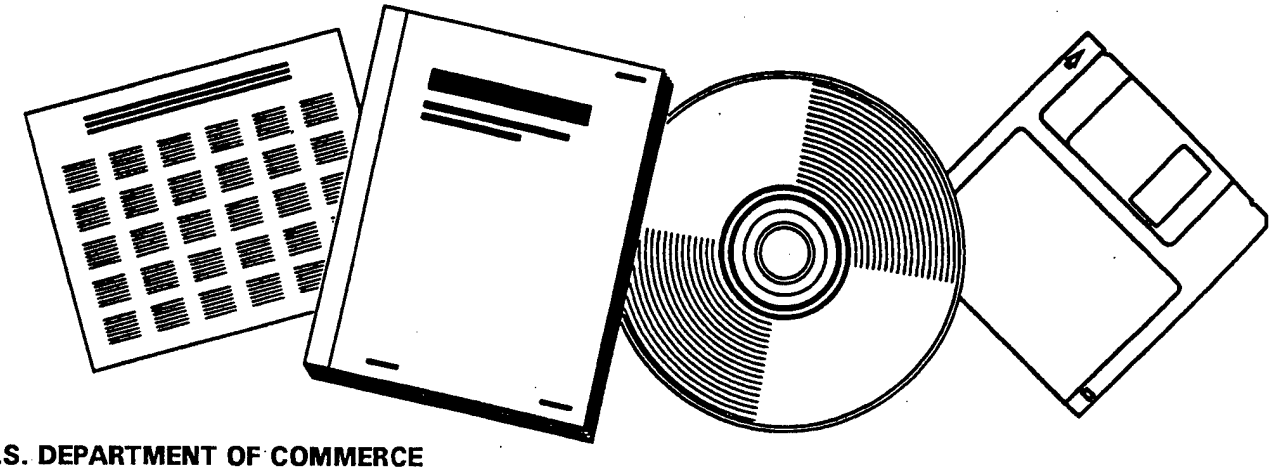
PB96-148812

DESIGN AND IMPLEMENTATION OF POLIGON A HIGH-PERFORMANCE, CONCURRENT BLACKBOARD SYSTEM SHELL

STANFORD UNIV., CA

NOV 89



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

November 1989

Report No. STAN-CS-89-1294

Also numbered as KSL-89-37

PB96-148812

The Design and Implementation of Poligon, a High-Performance, Concurrent Blackboard System Shell

by

James Rice

Department of Computer Science

Stanford University

Stanford, California 94305

Knowledge Systems Laboratory
Report No. KSL–89–37

November 1989

The Design and Implementation of Poligon, a High-Performance, Concurrent Blackboard System Shell

by
James Rice

(Rice@Sumex-Aim.Stanford.Edu)

Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA* 94304

The author gratefully acknowledges the support of the following funding agencies for this project: DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

---

* California constantly emits neutrons, which strike other materials and make them radioactive. — *Birmingham (Ala) News*

Contents

Table of Figures

Table of Drawings

Abstract

I started at the top and worked down.

— Orson Welles.

This paper discusses in detail the design and implementation of Poligon, a concurrent blackboard system, documenting our progress and the problem areas we identified in the process of developing it. It also considers the factors that aid and those that limit the performance of blackboard systems in general and of concurrent blackboard systems in particular, relating these factors to the implementation of Poligon.

1. Introduction

Six Hours a-Day the young Students were employed in this Labour; and the Professor shewed me several Volumes in large Folio already collected, of broken Sentences, which he intended to piece together; and out of those rich Materials to give the World a compleat Body of all Arts and Sciences; which however might be still improved, and much expedited, if the Publick would raise a Fund for making and employing five Hundred such Frames in Lagado, *and oblige the Managers to contribute in common their several Collections.*

— Jonathan Swift, *Gulliver's Travels, Chapter 5 of Part III* "A Voyage to Laputa"

The Advanced Architectures Project [Rice 88c] has already published a large number of research results, for example [Nii 88b] and [Saraiya 89]. Up to now, however, we have not described the actual *implementation* of the systems that were produced in order to do our research. During this research we identified solutions and potential problem areas for designing future systems in our target area of research, namely concurrent problem-solving systems. It is important to us that we should be able to disseminate the knowledge that we have gained through our experience so that the obstacles we encountered can be avoided by others. Thus this paper not only highlights our positive results, but also attempts to evaluate our approaches and the problems inherent in these systems.

In this paper we describe the design and implementation of Poligon [Rice 86], a concurrent blackboard system [Nii 86].[1] In this section we briefly outline the reasons why we built Poligon, in Section 2 we describe the design and implementation of AGE [Nii 79], perhaps the archetypal, serial blackboard system shell, so as to introduce the discussion of Poligon's design. In Section 3 we briefly consider the implications of the blackboard model for parallel execution. In Section 4 we discuss the design and implementation of Poligon describing in detail its internal representation. Section 5 focuses on the design of Poligon's program support environment. Throughout the paper we attempt evaluate our approaches based on the outcomes of the project, which are summarized in Section 6.

---

[1] It may be of interest to note that the name *Poligon* originated from the system's ability to run in both serial and parallel modes. Names of parallel systems often begin with the letter *p*; *Poligon* combines the Greek word Ὀλιγος (few) and Πολυγονος (producing many, prolific, from which we derive *polygon*, a many-sided object). Following the same pattern, Poligon's non-CARE mode is called Oligon.

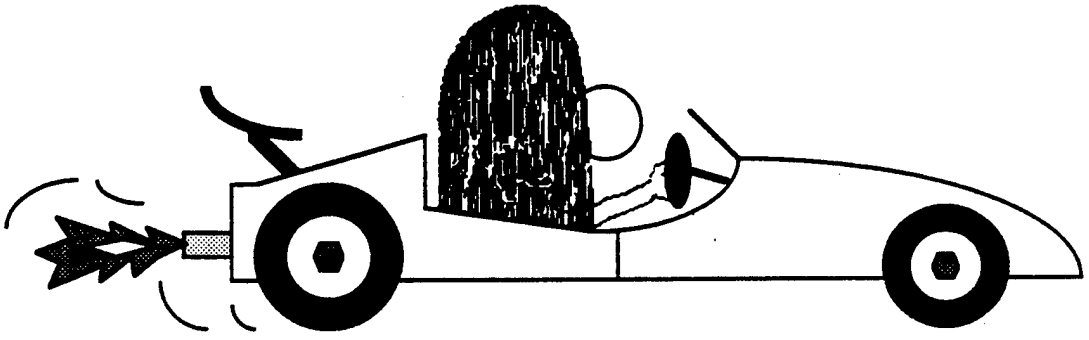
In this paper we assume at least a passing acquaintance with knowledge-based system shells and with concurrency, synchronization, critical sections, and other related concepts. Those less familiar with these issues are directed to [Nii 88a], which provides a thorough explanation of the issues and terminology involved.

This paper discusses not only Poligon, a system that we implemented, and AGE, a system that was implemented a number of years ago but also different ways in which Poligon or some future systems *could* be implemented. We have endeavored to distinguish these subjunctive systems from those that have actually been implemented, but the reader should still be aware that in order to state our beliefs and hypotheses about the future of concurrent blackboard systems we inevitably have to describe how we would implement Poligon in the light of what we have learned or what we would have done if our goals had been different. For instance, some design decisions were based on the goals of flexibility and ease of implementation, whereas if we were to implement again with the primary goal of peak performance, we might choose entirely different design and implementation strategies. The reader should, therefore, note phrases such as "future implementations might...", and "in the best of all possible worlds...," which indicate some impending speculation rather than statement of fact.

1.1. Why High Performance?

Quick is beautiful.

— F.J. Dyson



Eagar likes high-performance machines.

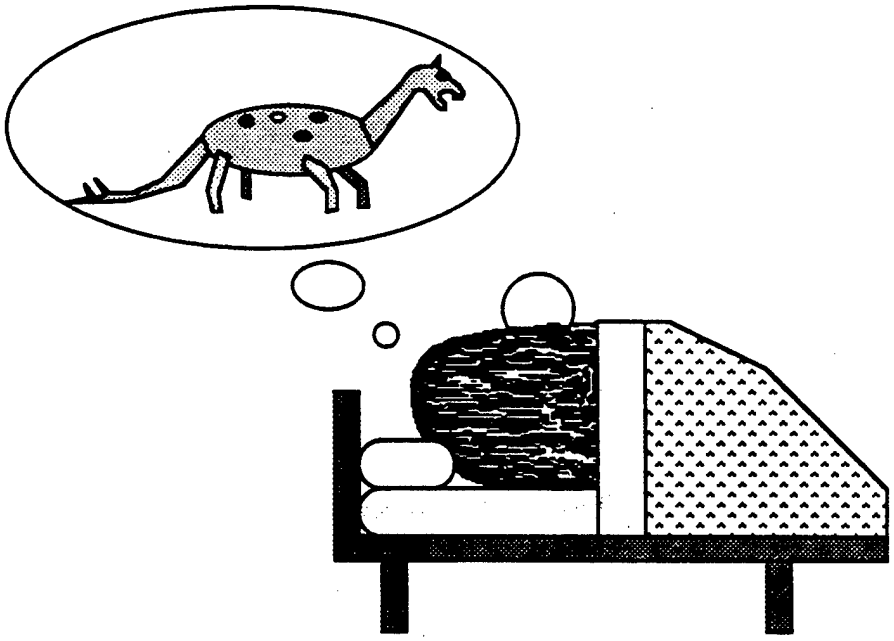
AI research has proceeded for some time without much concern about performance. Researchers have mostly been concerned about the behavior of their programs and were satisfied as long as their programs executed in reasonable time. Now that the technology is maturing and there is increasing pressure to apply AI programming techniques to previously intractable problems in the real world. This inevitably means that the performance of these systems must be compatible with the real world. Hearsay II [Erman 80] provides a good example of this. Even if it had worked perfectly, it would still, at that time, have operated at least ten times too slowly to have been used in the real world. The problem domain we chose to investigate was the interpretation of multiple, continuous signal data streams, such as one might find in radar systems. We already knew this domain to be one in which current blackboard systems have not been able to cope with the performance demands of the real world.

1.2. Why Concurrent?

A physicist had a horseshoe hanging on a door of his laboratory. His colleagues were surprised and asked whether he believed that it would bring luck to his experiments. He answered: "No, I don't believe in superstitions. But I have been told that it works even if you don't believe in it."

— I. B. Cohen

To begin with we should address the question of why we are looking at concurrent systems at all. As mentioned earlier, we need more performance from our AI software in order to apply it to real-world problems. To accomplish our goal, we also need to develop programming methodologies to help us use the evolving generation of machines. It was anticipated that the use of multiple processors could deliver the desired increases in speed. Thus, we wanted parallelism solely in order to gain performance, not to model the physical separation of processors in a distributed, multi-agent system or to improve the reliability of our software.



Eagar likes to be Parallel.

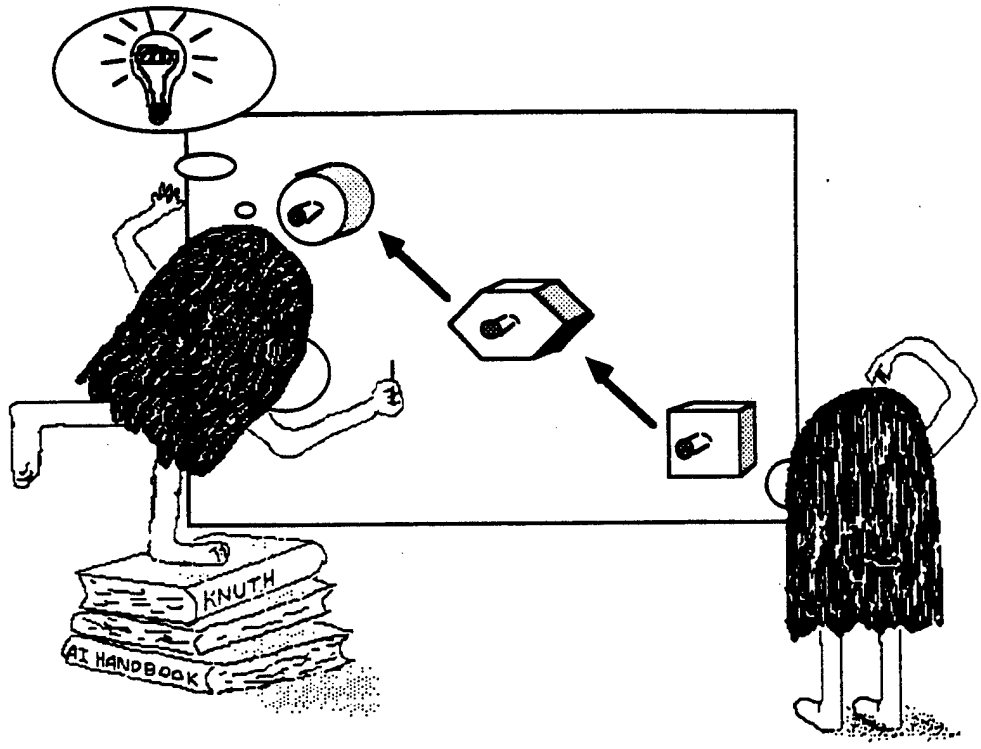
1.3. Why a Blackboard System?

On the atomic bomb: *That is the biggest fool thing we have ever done. The bomb will never go off, and I speak as an expert in explosives.*

— Admiral William Leahy to President Truman (1945).

Having decided that to investigate concurrent systems, we had to determine which software architectures we were interested in. The primary question was: *Why not write everything in C or assembler with suitable parallelizing directives?* This is by no means a trivial question. The development of any high-level programming tool is based on the hidden assumption that its benefits outweigh its costs. In choosing some form of high-level programming tool over a low-level programming tool, the trade-off is hard to justify when the

goal is a high performance system. Except for truly enormous systems, it is generally the case that software written in assembler is faster and smaller than software written in high-level languages. What one trades off, then, is greater ease of program development, modification, and maintenance against performance. The human cost of software development is great enough that it pays to spend more money on hardware to get the required performance than to spend money on the software being developed.



Eagar finds that a blackboard helps him organize many experts.

Generally this argument applies only in areas of specialized software. Word processing software, like that used to write this paper is usually sold in such quantities that despite the high cost of programming, it is worthwhile to develop new software for existing platforms using less productive methodologies that result in faster program execution. There is, however, a large domain of applications that are only run on a few machines. This software must be developed quickly and modified and maintained easily. Nowhere is this more apparent than in the development of AI software.

Thus, what we are saying is that when we commit ourselves to speeding up expert-system applications using of parallel hardware. We are committed to designing software that can meet its intended purpose. If we elect to design a low-level tool, we accept that it may be hard to use, but it must be very fast. If we design a high-level tool then not only should it be able to solve the problems reasonably quickly, but it should also deliver the benefits that are claimed for high-level tools; modifiability, maintainability, and speed of code development. We were more interested in the design of high-level tools so we were compelled to develop software architectures with the capability to handle the rapid development of concurrent expert systems while still giving high performance. To do this, we sought a computational model around which to develop our design.

At the time, the most promising contender for our prototypical software architecture was the blackboard architecture. Our experience in using this architecture on our project was a significant advantage, but in addition to this, the design itself seemed to admit parallelism through being an intrinsically concurrent problem-solving model. It also seemed to meet our need for a high-level computational model that would help the programmer deal with the complexity of future AI systems. We later learned that blackboard systems are not as parallel as we originally thought; why this is the case is documented in fair detail in [Rice 88a]. Our example suggests, therefore, that one should not pick a programming model for reasons of a superficial match to one's cognitive model of concurrent problem solving. Many of our findings were considerably at variance with our intuition when we started the project. We know of no better architecture than the blackboard model for concurrent problem solving, but this may simply be that few have tried others, other than simple production systems [Gupta 86].

The rest of this paper is biased toward the design of blackboard systems; however, a number of the lessons we learned have broader applicability than just to the field of blackboard systems. Because the blackboard programming model has achieved considerable popularity for reasons independent of its performance, it is quite likely that many will attempt the implementation of concurrent blackboard systems and can benefit from our experience.

2. The Implementation of an Existing Blackboard System - AGE

> *What we want is a story that starts with an earthquake and works its way up to a climax.*
>
> — Samuel Goldwyn

In this section we discuss the design of AGE, a blackboard framework developed at Stanford, both to provide historical and technical background, and to help elucidate the issues involved in developing our project goals.

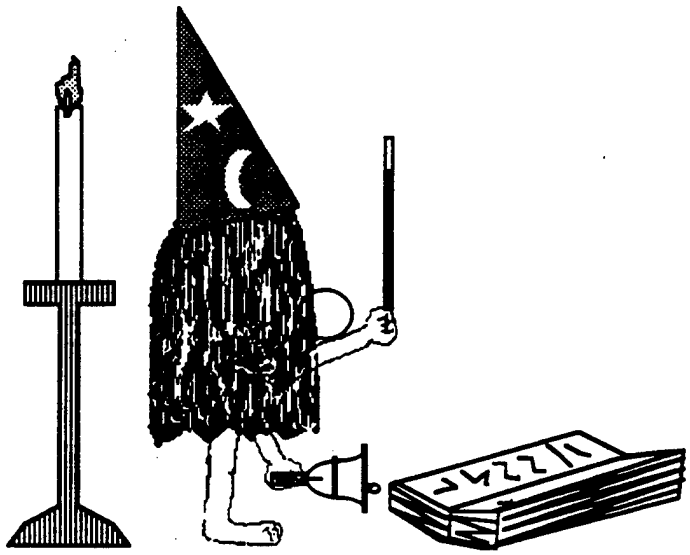
AGE is a blackboard framework written in Interlisp. It is significant that it is a framework. There have been a number of hard-coded blackboard systems, HASP/SIAP [Nii 82] and Hearsay II [Erman 80] being the best known. We were not interested in the development of hard-coded solutions to our applications since this would have violated the trade-off mentioned in Section 1.3. Having developed the tool, the marginal cost of developing more applications should be relatively small. This is, of course, the same argument that led to the development of compilers.

AGE is a system whose design is geared toward the rapid development of blackboard applications. It provides a toolkit for blackboard system development, which contains the infrastructure in which the user's knowledge is to run, as well as such things as rule editors.

2.1. The Blackboard Model

Although the main purpose of this paper is not to explicate the blackboard programming model, it is useful to give a brief description of a canonical blackboard system, in order to show how AGE implements this model. For further information on various blackboard systems the reader may wish to consult [Engelmore 88].

In the blackboard problem-solving model, a group of experts is gathered around a blackboard, each contributing his own knowledge toward solving the problem at hand. The experts communicate by posting conclusions on the blackboard and watching for other experts posting their conclusions in a similar way. When an expert spots a piece of information that he knows how to handle, he starts working with it. By this means the solution evolves.



Invoking a knowledge source.

This problem-solving model cannot be immediately implemented for a number of reasons, but the primary change that is needed to turn this problem-solving model into a programming model is the inclusion of a scheduling mechanism. This is often referred to as an *opportunistic scheduling scheme* and is often thought to be central to blackboard systems, even though it is only a product of their *implementation*, rather than their *design*. In this case, *opportunistic* means that *the system is sensitive to changes within the evolving solution and, in some manner, tries to invoke the most appropriate piece of knowledge at any given time in order to help the progress of the solution.* This is in contrast to conventional operating system scheduling models in which the scheduler itself has no knowledge of the intent or importance of any given process other than through the use of some "magic" numbers such as priority or quantum numbers. Clearly, a good knowledge-based scheduler ought to be able to use knowledge of the application domain and of the knowledge being executed to find a more responsive and efficient scheduling order.

2.2. AGE, the Canonical Blackboard Shell

Titus Lartius: *Follow Cominius; we must follow you;*
Right worthy you priority.

— Shakespeare, *Coriolanus*, act I scene I.

In this section we discuss the implementation of AGE, highlighting the factors governing its performance.

AGE, being a blackboard system, has a global database that is used to represent the evolving solution – the blackboard. This database is implemented within the native Interlisp en-

vironment's heap. The blackboard is made up of a number of data structures that represent the different elements in the solution space. These solution-space elements, called nodes, contain mappings from user-defined names to the values they represent. For instance, a node might have a slot called *parent*, which has as its associated value the parent of the node in question. These mappings are usually referred to as *attribute/value pairs*.

The knowledge base is composed of a collection of knowledge sources (KSs). These are structures that contain a set of rules that are applied when a knowledge source is invoked. The code for these knowledge sources is also resident within the Lisp system's heap.

A typical blackboard application written in AGE has the following behavior and is shown in Figure 2-1.

- Data coming into the system results in the creation of nodes on the blackboard. These nodes have their slots initialized so that they have some meaningful values in them.

- An event token is passed to the scheduler; in turn the scheduling mechanism invokes the knowledge sources that are interested in that type of event. This involves searching the knowledge base for applicable knowledge sources.

- During the invocation of a knowledge source, computation is performed in order to construct some context relevant to that particular invocation of the knowledge source. The named components of the context are referred to as *knowledge source bindings*. These take the place of local variables in knowledge sources and map local identifiers into computed values. Once these values have been computed, the rules attempt to fire. Rules are implemented as condition/action pairs. If the condition is true, the action or actions are invoked.[1]

- Clearly, the evaluation of knowledge source bindings and of any expressions within rules and knowledge sources must be able to look at the nodes on the blackboard. If this were not the case, the knowledge represented by the knowledge source would be unable to do any computation that was dependent on the state of the solution. For this reason, AGE supports a function (called $Value) that will read the value or values associated with a particular attribute on a particular node. This is AGE's slot read operation.

- Similarly, the knowledge in the system must have some way to record its conclusions. This is done in one of two ways: either the rule that is executing will modify a node or nodes on the blackboard to conform to its new model of reality, or it will create new blackboard nodes to represent new parts of the solution.

- Finally, having performed any appropriate side-effects on the blackboard, the AGE application must perform some action in order to make sure that the system notices the changes that have been made.[2] This is done by naming the changes with an

---

[1]AGE also supports a mechanism for selecting which rules within a given knowledge source are to fire, but this is not germane to our discussion here.

[2]In the first implementation of MXA [Rice 84], every modification to every slot generated an event. The number of events to be processed grew so large that the application was not able to deal with them reasonably. AGE's strategy of leaving the posting of event tokens to the user is a less automatic approach but more reasonable in practice.

event token. It is this event token that the scheduling mechanism sees in later system cycles and that causes the subsequent invocation of further knowledge sources.

- The system loops around, looking for events on the event queue and processing them in the manner described above. The process of acting on an event and looping around to process the next event is referred to as the *system cycle*.

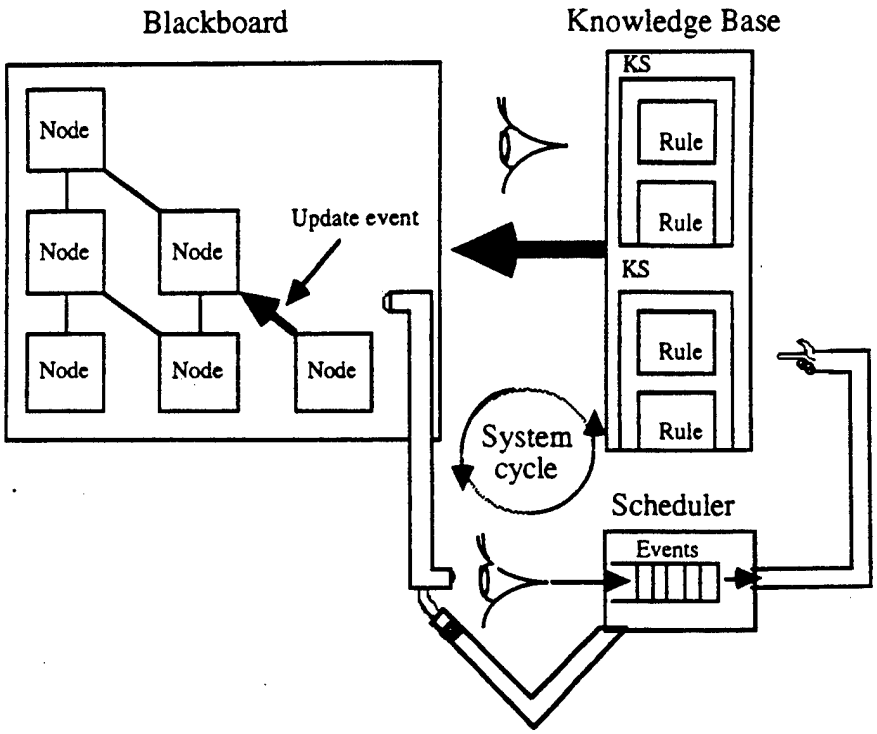


Fig. 2-1. This figure depicts some fundamental aspects of most blackboard systems. A central scheduler sees or is informed of changes on the blackboard, noting them in an event queue. Events are selected from this queue and are used to trigger knowledge sources, which in turn act on the blackboard.

Now that we have outlined the essential run-time behavior of AGE, we can distill from this description the essential components of a blackboard system. These are:

- Dynamic node creation
- Knowledge search — finding applicable knowledge sources for a given event type
- Conflict resolution — deciding which knowledge source to invoke if more than one is currently invokable
- Knowledge invocation — firing up a knowledge source once it has been selected
- Context evaluation — evaluating any user code necessary for the knowledge source
- Slot reads
- Slot updates — the side-effects that propagate conclusions
- Event posting — recording that something significant has happened

It should also be noted that many blackboard systems have a mechanism for finding nodes that match a certain predicate, AGE's $Find and MXA's [Rice 84] set creation mechanism are but two examples.

The relative importance of these different aspects of a blackboard system will depend very much on the architecture and on the application for which it is being used. For instance, a system with a large knowledge base of simple rules will stress the knowledge search and invocation mechanisms; an application that does a lot of raw number crunching will require the rapid evaluation of user code. It seems likely that any blackboard system implementation tool (shell) must have some means of performing these tasks, and if it has any aspirations to high performance, a reasonable strategy for making them efficient.[1]

AGE was designed primarily as an experimental tool and so was optimized more for program development than run-time performance. We will now discuss the implementation of each component of AGE so that we can contrast its implementation with that of Poligon.

Node creation. AGE nodes are implemented as slots on the property lists of the symbols that name the nodes.[2] The slots within nodes are represented simply as an AList. Thus, the instantiation of nodes simply involves the creation of the data structure and the recording of it in the level (class) of nodes of the same type. A consequence of this architecture is that nodes of a given level — aircraft, for instance — are only similar by convention. Any node can have a collection of slots that is totally different from another node that is notionally of the same type. This means that space will, in principle, not be wasted in nodes that never use certain slots.

Knowledge search. AGE, like many blackboard systems, offers a user-programmable scheduling mechanism with various precanned strategies. By default events are selected from a global event queue in AGE. Each event encapsulates both the node that caused the event and the event token, which is used to select the applicable knowledge sources in the next cycle. The event token is compared with the preconditions on each knowledge source in the knowledge base, and the set of applicable knowledge sources is delivered. The knowledge source precondition merely has to name the event token against which it is to match. This precondition can be thought of as a filter that helps to select potentially applicable rules.

Conflict resolution. AGE's conflict resolution strategy is extremely simple. If more than one knowledge source could be triggered from an event, then the matching knowledges are fired in the lexical order of their definition.

Knowledge invocation. In AGE, knowledge sources are implemented as list record structures that are interpreted by the scheduling mechanism. The triggering node (the focus node) is taken from the event queue and dynamically bound to a global variable, called *focus.node*. During the invocation of a knowledge source, code executes any knowledge source bindings and then attempts to fire

---

[1] By way of qualification, we should say that at present most serial blackboard systems are optimized for executing either *search* or *recognition* types of applications. It would perhaps be unreasonable to expect any different from a parallel system, though in the best of all possible worlds a blackboard tool would be good at both of these tasks.

[2] A unique identifier is CONSed to name each node.

the rules by successively testing their conditions and executing their actions, if appropriate.

Context evaluation. In AGE, all user code is interpreted. This means that all knowledge source bindings and all expressions that are evaluated during the execution of rules are also interpreted.

Slot reads. The $Value function performs Slot reads in a regular manner. It can read the value of slots on any node on the blackboard with cost independent of the node being read. The $Value function must access the AList within the node structure and must then search for the value named by the slot being accessed. This search must be performed because even if the system were to be compiled it would not be possible to establish at compile-time the location of any given slot within any given class of node.

Node updates. AGE provides a fixed number of system-defined ways to update a node. These allow the modification of the value lists associated with a collection of slots and are performed by calling a procedure ($Modify or $Supersede) with a set of arguments that are interpreted so as to find the slots to be updated and the values to put into those slots.

Event posting. Event posting is simple in AGE because of its centralized event queue used by the scheduling mechanism. Whenever an event is to be posted, AGE invokes a procedure that encapsulates the node causing the event and the event token and pushes the event onto the front of the event queue.

Search. Searching a blackboard in a serial system with most implementation techniques is likely to be a linear time operation at best and highly combinatorial at worst. AGE's $Find operation searches linearly through all the nodes on a blackboard level for a match.

What a high-performance blackboard system should do, therefore, is find ways to make each of these operations fast while preserving the blackboard model.

3. Implications for Parallel Systems

Let's bring it up to date with some snappy nineteenth-century dialogue.

— Samuel Goldwyn

The AGE blackboard model discussed above is based on a number of hidden assumptions that preclude parallel execution. In this section we touch on some of these issues in order to show why certain implementation decisions were made in Poligon.

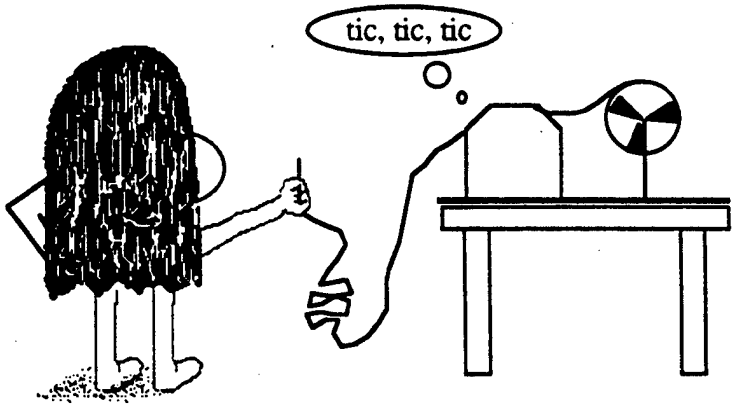
3.1. The Right Answer

> A second order approximation for evaluation of the Köchel (K) numbers of a Mozart symphony (S): $S = 0.27465 + 0.157692K + 0.000159446K^2$
>
> — R.P. and J.R. Cody[1]

We should first stress that a concurrent implementation should get the right answer. This is not at all a trivial point and bears some thought. In serial blackboard systems such as AGE, only one thing ever happens at once; in other words, there is no ambiguity about the degree to which the system is converging toward a solution. It is possible to construct a concurrent problem-solving architecture that will act identically to a serial system, but the amount of synchronization required is great enough that the parallel implementation is likely to be slower than the serial version because of the costs of process creation, process switching and synchronization.[2] As soon as one starts to relax the requirement that a concurrent system should have the same semantics as a serial system, the nondeterminism so introduced can result in a system that either behaves quite unpredictably or fails to converge towards any solution at all.

As a corollary we can say that because a concurrent system allows the simultaneous investigation different avenues that may lead to solutions of differing quality, it is possible to trade off the accuracy of results produced against the overall performance of the system.

Eagar-Jones Average	
Pork Belly Futures	$4.23
Condo Cave Dwellings Inc.	$0.03
Wife-Grabber Clubs Corp.	$9.42
McBrontoburger Intl. (Franchize) Inc.	$3.11



Eagar finds that timing can be crucial to getting the right answer.

---

[1]This method will give an answer not more than two out, 85 percent of the time.

[2]This need not be the case if the serial program is, for instance, a pure applicative program, the semantics of which are identical when executed in parallel, but this statement holds true for the parallelization of most current serial programs.

3.2. Instances and Processes

Dogberry: *Come, bind them. Thou naughty varlet!*

— Shakespeare, *Much Ado About Nothing*, act IV, scene II

In concurrent systems it is frequently the case that in order to implement concurrency, each piece of concurrent computation must be executed within a process. For a number of system implementation reasons, these processes are often large and expensive. A serial system need not worry about such matters. Even if such things as dynamic binding are eliminated from the system, process switch time is still likely to be substantially greater than the native system's function call overhead because of the cost of reloading caches. In addition, the cost of processes has a substantial impact on the programming model. This is because an appealing programming model for concurrent computation is that of asynchronously communicating objects. In medium-grain-sized machines, these objects are generally tens to hundreds of bytes in size. Because of page-based stack protection hardware and the increasing size of pages in modern machines, even the minimum size of a stack group is likely to be tens or hundreds of times that of the objects in the system. This means that one cannot sensibly allocate a process to each object without either accepting a huge loss in memory performance or choosing some architecture or computational model that makes more efficient use of stack groups.

3.3. Data Types

Mathematics are a species of Frenchmen; if you say something to them, they translate it into their own language and presto! it is something entirely different.

— Göethe.

Existing serial systems have a well-understood set of data types that are geared toward both the efficient use of existing hardware and the implementation of programmer abstractions. An example of this is structure types, which are often implemented as arrays. Array indexing is fast on all machines. Thus, the user is guaranteed the efficient implementation of his program while preserving the abstraction of naming fields in data items symbolically.

It is not clear yet whether these data structures are appropriate for general concurrent computation, let alone AI programming. CMLisp [Hillis 85] is an example of a language in which new data structures are used to enhance parallelism. Certainly researchers will have to think hard about what data structures are appropriate for concurrent problem solving. Once a reasonable consensus has been reached, we must then convince hardware implementors that these new data types should be supported efficiently in their hardware. Poligon made some steps in this direction, as is mentioned in Section 4.10.

3.4. Control

Control, or *MetaKnowledge* as it is often called, is intrinsically a serializing process, at least as we understand it in the serial blackboard sense. This is because the act of stopping to decide what to do next requires synchronization and then serial processing of the decision process, followed by the serial execution of the knowledge that is selected. Similarly, the knowledge that decides to post events must synchronize on the shared event queue. Strong evidence to support the assertion that control is intrinsically serializing is given in [Nii 88a] and [Aiello 88].

To make efficient use of parallel processors, therefore, a concurrent problem-solving system must try to find ways to avoid the overhead associated with scheduling. As a consequence, system performance will probably degrade because the system is unable to apply the best knowledge all of the time. But given good design, one can at least hope that saving the cost of control and the parallelism extracted as a consequence will buy back by many times the loss in performance caused by executing suboptimal knowledge.

3.5. Hardware

The degree to which our normal serial programs match the hardware on which they run is something we all take for granted. The languages in which we express the programs are themselves biased toward the efficient use of the hardware and vice versa. This is less likely to be the case in the near future. New programming models will have to evolve to cope with new hardware designs, and new programming methodologies will have to be developed. It is clear that a good match between the granularity of the hardware and that of the program will be crucial to the efficient execution of user programs. Likewise, it may well be the case that a good match between programming model and memory architecture will be required. Programming models that use message passing may well be the best application for distributed-memory message-passing hardware. A shared-variable programming methodology may make more efficient use of shared memory machines. This is discussed in [Byrd 88] (see Figure 3-1). An example of a concurrent blackboard system designed to operate on shared-memory machines is Cage [Aiello 86], also part of the Advanced Architectures Project.

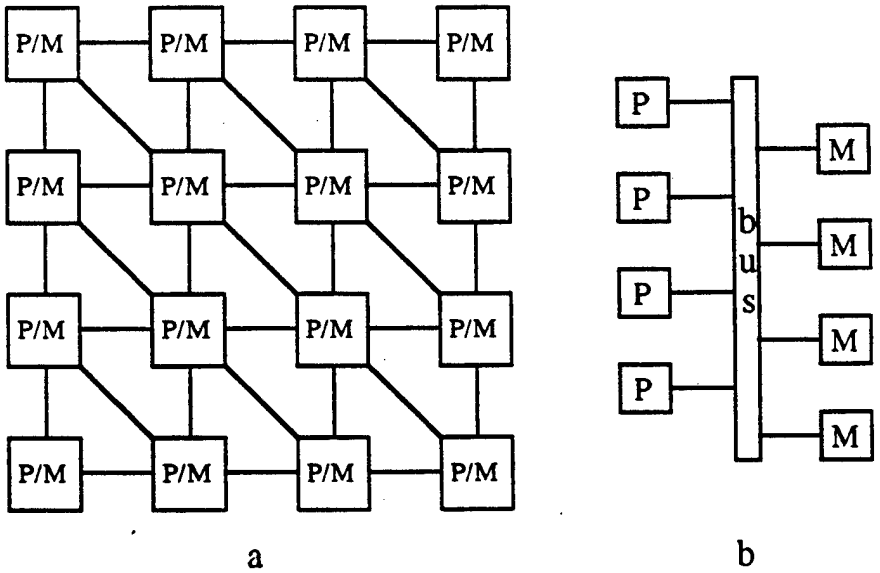


Fig. 3-1. a. A distributed memory machine consists of a collection of processor/memory (P/M) pairs linked by some network — in this case, a six-way connected array. b. A shared memory machine consists of a collection of processors that view a collection of memories as a global resource. In this case, a bus connects the processors to the memories.

3.6. Real-Time

Real-time systems have some special attributes that must affect our way of thinking in a parallel computational environment. Data is likely to arrive out of order if, for instance, network congestion causes unpredictable delays in message transmission. Therefore, pro-

grams must be rugged with respect to data being garbled and must be able to do the "right thing" even when different parts of the program are working at wildly different rates.

4. The Implementation of Poligon

Basic research is what I am doing when I don't know what I am doing.

— Wernher von Braun

In this section we discuss the implementation of Poligon in detail, discussing its history, the problem areas we encountered and, in particular, the areas of a blackboard system mentioned in Section 2 that we believe require improvements in efficiency.

Our initial ideas about Poligon were strongly influenced by the primary objectives of the Advanced Architectures Project. It was broadly assumed that silicon was going to be cheap. We could afford to produce a resource-inefficient design as long as it was usefully faster than one that was more resource efficient. Similarly, we assumed that our hardware design, which was to be run in simulation, would be strongly driven by our evolving programming models. This would allow us to assume the existence of a "blackboard machine" and thus produce designs that could not be efficiently implemented on existing hardware, but that could be implemented on a blackboard machine with suitable hardware or microcode support.[1]

These assumptions proved not to be valid simply because of the way the project developed. The hardware design component of the project progressed at a greater rate than the software design, and by the time, some of our problem-solving software began to be implemented, it was clear that we would have to reconsider some of our design decisions in order to get an efficient implementation on the hardware that had been designed. Clearly lack of experience of the real problems of concurrent programming may well have marred our early decisions. The reader is therefore advised to view the following description of the evolving design of Poligon in terms not only of increasing understanding of the underlying problems but also of a gradual appreciation that the targets that we thought were fixed at the beginning of the project were, in fact, moving.

An overriding consideration in the design of Poligon was to develop a system that could, at least in principle, be highly compiled. Existing systems usually have to rely on a great deal of interpretation. Consequently, it was decided early on that if we wanted a feature x to give us the functionality that already exists in serial blackboard systems and there was a similar feature x' that gave similar functionality, but was more highly compilable, we would choose x' over x. This philosophy strongly influenced the designs described in this section.

4.1. The Programming Model

I had a good idea this morning but I didn't like it.

— Samuel Goldwyn

During the initial design of what later became known as Poligon, we decided that we wanted to preserve the abstraction model provided by the blackboard programming

[1] It is not at all clear, of course, whether anyone will ever build a dedicated blackboard machine of this type.

metaphor. We already suspected that control would be a significant serializing factor and thought that the communication path between the knowledge base and the blackboard would be a bottleneck if we naïvely parallelized a serial blackboard system. We resolved to produce a design that would eliminate these factors as much as possible, while still retaining the characteristics of a blackboard system.

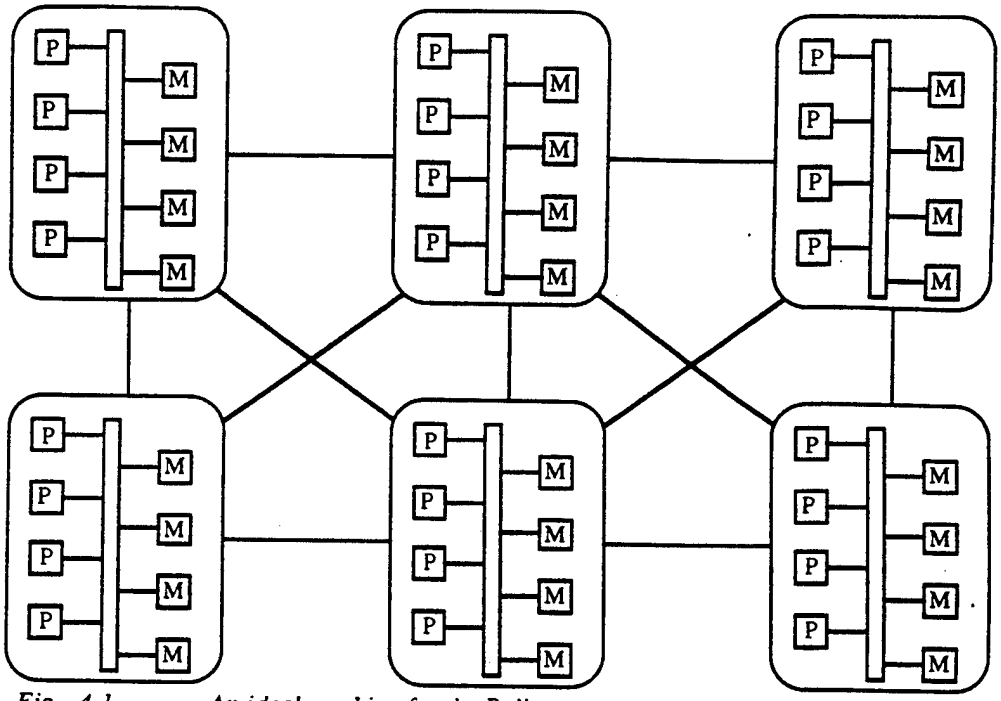


Fig. 4-1. An ideal machine for the Poligon programming model would probably be a collection of shared memory machines linked as if they were a distributed memory machine. This would allow tight coupling and the sharing of data between rule invocations for a particular node and efficient loose coupling between the nodes on the blackboard.

We decided to make the nodes on the blackboard into active agents. A compiler would attach relevant knowledge to the nodes at compile/load time, and the nodes would then invoke their knowledge as daemons triggered by changes to the nodes. This meant that all centralized control would be removed and that all relevant knowledge would have direct access to the data with which it was most concerned. These design ideas were strongly influenced by arguments from our hardware developers that suggested that multiprocessors having very large numbers of processors are most likely to be distributed memory machines. As a result, the cost of reading data from a local processor's memory would probably be much less than that of reading it from a remote processor/memory pair, at least until appropriate new programming models could be developed. It seemed that our own programming model should in some way reflect this asymmetry, though we initially hoped that we could shield the Poligon programmer from this. An idealized machine model for the Poligon programming model is shown in Figure 4-1.

It is important to note that the programming model was strongly influenced by the known implementation model of the CARE machine. This model encouraged a value-passing model of computation, so Poligon was to allow no global variables, and the values transmitted as arguments to any messages sent by the system would be copies of the original values, not remote pointers to the actual values. Remote-Address pointer objects were the only type of pointer that could be transmitted between processing elements. These are

pointers to the streams used to communicate between processes. In the CARE machine, the copying of data is performed by a special processor that handles operating system and communication functions. The user's application is not held up by the copying of message arguments, since this happens in parallel with user code evaluation. Thus, in the following discussion, whenever reference is made to messages being sent or to values being transmitted the reader should remember that these are always copies of the data structures on the originating processor.

Another aspect of the CARE machine model is the semantics of message passing. Unlike the sending of a messages in a Flavors program, for instance, messages in the CARE machine model do not have procedure-call semantics. Returning a value from the computation performed as the result of a message is not mandatory, nor, when a value is returned, will this reply necessarily be sent to the originator of the message. Messages in CARE have explicit *clients*. The *clients* of a message are a collection of the nodes that will need to know the values derived from the computation invoked by the message. This set may be null. Therefore, while much message passing in Poligon has procedure-call semantics this is only because the clients of the messages are often the same as the originators of the messages. This is not always the case, however.

4.2. The Structure of Nodes

> On trapping a lion in a desert [Petard 38]: The "Mengentheoretisch" method. *We observe that the desert is a separable space. It therefore contains an enumerable dense set of points, from which can be extracted a sequence having the lion as limit. We then approach the lion stealthily along this sequence, bearing with us suitable equipment.*

The development of Poligon started on Symbolics™ Lisp Machines[1] and later, upon their arrival, continued on Explorer™ Lisp Machines.[2] Because of the strongly object-oriented programming model we envisaged for Poligon, we decided to implement Poligon using the native Flavors system resident on both of these MIT-based Lisp Machines. This decision was motivated primarily by a desire for good performance, compatibility, and good support from the programming environment. Considerable programming effort had already been spent on the development of the CARE simulator [Delagi 88a], which is also written in Flavors. This encouraged us to keep a homogeneous implementation with the underlying simulator.

Poligon nodes, therefore, are implemented as instances of Flavors. We had decided to trade extra compilation effort in favor of higher performance, so we were able to implement slots using Flavors instance variables. We knew that the Flavors model itself would only be adequate as a low-level implementation model. Since the message-passing semantics of Flavors programs are incompatible with the message-passing semantics that we envisaged from the simulated hardware, we had to build a number of layers on top of the Flavors representation of nodes. For consistency the classes of nodes on the blackboard were themselves represented on the blackboard. This was a departure from the AGE model, in which the levels were not really on the blackboard as first-class citizens. In Poligon it was decided that classes should be first-class citizens, and that we should have a general class/metaclass hierarchy in order to describe the complexity of the taxonomy in the problem domains we envisaged and to implement Poligon's equivalent of class variables. The benefits of multiple compile-time inheritance also seemed worth having in Poligon.

---

[1]Symbolics is a trademark of Symbolics Corporation.
[2]Explorer is a trademark of Texas Instruments, Inc.

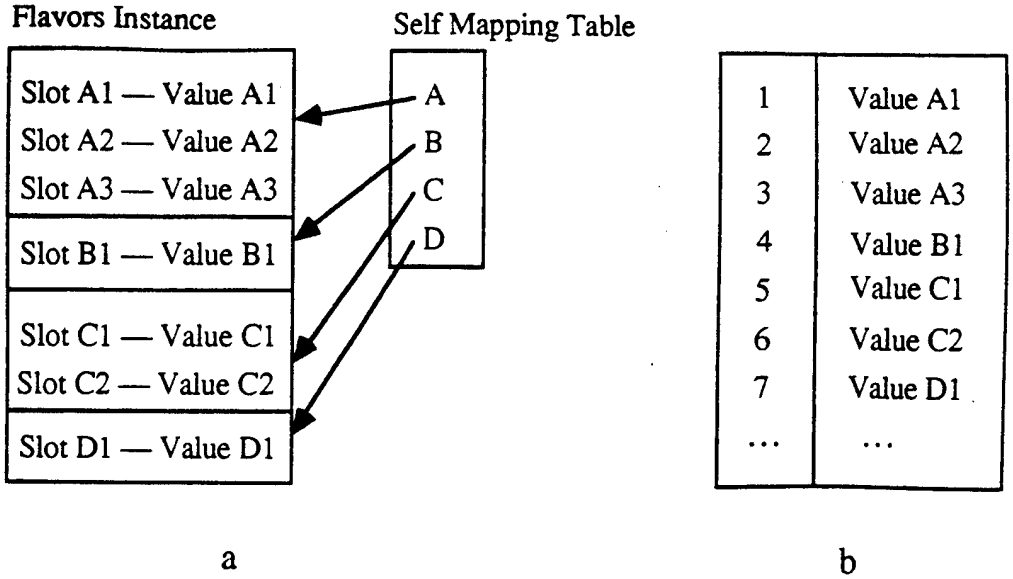


Fig. 4-2. a. The implementation of nodes in Poligon as Flavors instances. To find a slot, the system must indirect through the Self-Mapping Table to find the offset of the component flavor in the instance. b. An ideal implementation of nodes in Poligon would compile all slot references into array indices.

We knew from the start that, in an ideal real-world implementation of the Poligon model, a high performance blackboard system would compile its nodes into arrays, with slot references being compiled into simple array references. This was not done for ease of implementation. In a blackboard system, such as Poligon, however, one can trade off *some* generality for performance and allow optimization through somewhat strong typing and can make a simplifying assumption about the multiple inheritance on the blackboard. If the only classes that are ever instantiated are the leaf classes in the class hierarchy, then knowing the type of a node will always allow the computation of a slot as a fixed offset (see Figure 4-2). This implementation strategy would limit modularity, since it would not allow one to optimize rules that were inherited from abstract classes, but this might be a reasonable assumption in an implementation used in the field. Even without making this assumption, slot access can be effectively optimized given the type of the node in question, so this implementation seemed reasonable. Certainly, Poligon as we implemented it was not as well optimized as this, but at least in principle it could have been.[1]

Nodes, therefore, are instances of Flavors. These are composed in a set of class declarations specified by the user. The user now no longer has the ability to associate arbitrary

[1]Multiple inheritance can also be supported with fixed position slot access by the use of block compilation and a graph-colouring algorithm to allocate unique slot locations to all of the slots in the class hierarchy that is to be instantiated. Using this strategy, however, instances can easily end up with large "holes" in which slots for unincluded mixins could have been. The optimization of this method so as to minimize the size of these holes is a non-trivial problem but can have reasonable solutions for any given application. With this method, data space is traded-off against speed, whereas the strategy mentioned above trades off generality against speed.

properties with arbitrary nodes in the solution space. .. clear trade-off between run-time performance and space.[1] An example of this composition of classes is shown in Figure 4-3.

```
Class Flying-Thing :
    Slots :

Class Aircraft :
    Superclasses : Flying-Thing
    Slots :
        Wheels
        Wings

Class Bird :
    Superclasses : Flying-Thing
    Slots : Weight

Class FAA-Controlled-Thing :
    Slots : Serial-Number

Class Civil-Aircraft :
    Superclasses : FAA-Controlled-Thing, Aircraft
    Slots :
```

Fig. 4-3. Some example class declarations for a Poligon program. Birds and aircraft are flying things, and civil aircraft are both generic aircraft and things controlled by the FAA. Classes with names specified after the keyword Slots *are the names of slots added by the class, to which they belong.*

Nodes in the Poligon model communicate by posting messages to one another. These messages are not seen at the language level. Messages are received in a task queue and are processed one at a time by the nodes to which they were sent. Each node has its own such message queue, which is implemented as a stream (see Figure 4-4). Streams of values are one of the interprocess communications primitives that the CARE architecture supports. Objects running on a CARE machine communicate through these streams, and it is common practice for these objects to have only one stream that receive messages – the *self-stream*. In fact, whenever a Poligon program refers to a node, it is actually referring to the remote address of that node's self–stream. It is these remote addresses that are embedded in user data structures and passed around between nodes and processors.

As mentioned above, it is important for a concurrent programming model to have an efficient method for using stack groups. When we started the development of Poligon we had no such method, believing that nodes would not be all that expensive, and associated a fully fledged process with each one. The Poligon model for reading remote values was based on that of futures [Halstead 84]. This programming model, at least in its general implementation as used by Poligon, assumes that any process can stop at any point in order to wait for the value associated with a future, i.e., in order to perform the defuturing coercion. This may not be a good idea in practice because there can be pathological cases that use up all available memory by creating processes.

[1] One could generalize this by allowing behavior like si:property-list-mixin as well as fixed position slots, but we had no great interest in doing this for our applications. Clearly, slots of this type could not be as highly optimized as positional slots.

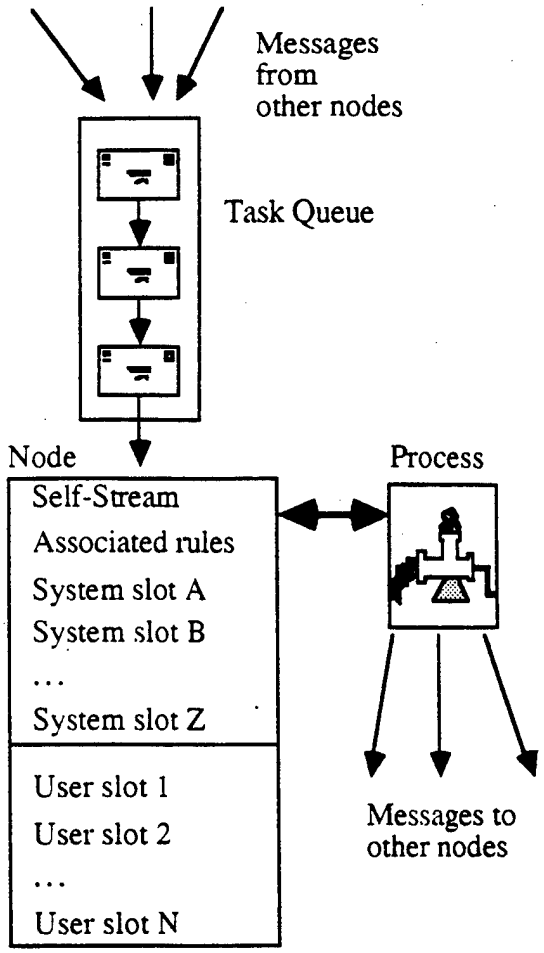


Fig. 4-4. Messages sent by nodes arrive in the node's self-stream. The node's process then processes them one by one, which may result in the sending of other messages.

Other work on the Advanced Architectures Project has developed a different programming model that is implemented in a system called *Lamina* [Delagi 86]. This model has restartable, run-to-completion code fragments. If a process needs a value from a stream that is not available, it aborts itself and restarts when a value arrives on that stream. This means that the process need not hold any state on the stack, so the stack group can be reused even though a process switch has occurred. There is clearly a need to be able to pass state onto the process when it is restarted so as to encapsulate the computation at the point of suspension. This is done by creating a closure that represents the continuation for the computation. The performance trade-off here is between the size of the heap-allocated closure that is CONSed, which will eventually have to be garbage collected, and the stack allocation of state, which is cheap while the stacks themselves are expensive. The programming trade-off is between the user being forced to encapsulate state explicitly and state being recorded automatically.

The advantage of the Lamina programming model is that it allows the user to have a good idea of the stack resource requirements of his programs. The disadvantage is that the user has lost the simple procedure-call semantics of a futures-based programming model.

4.3. The Rule-Triggering Mechanism and the Use of Stack Groups

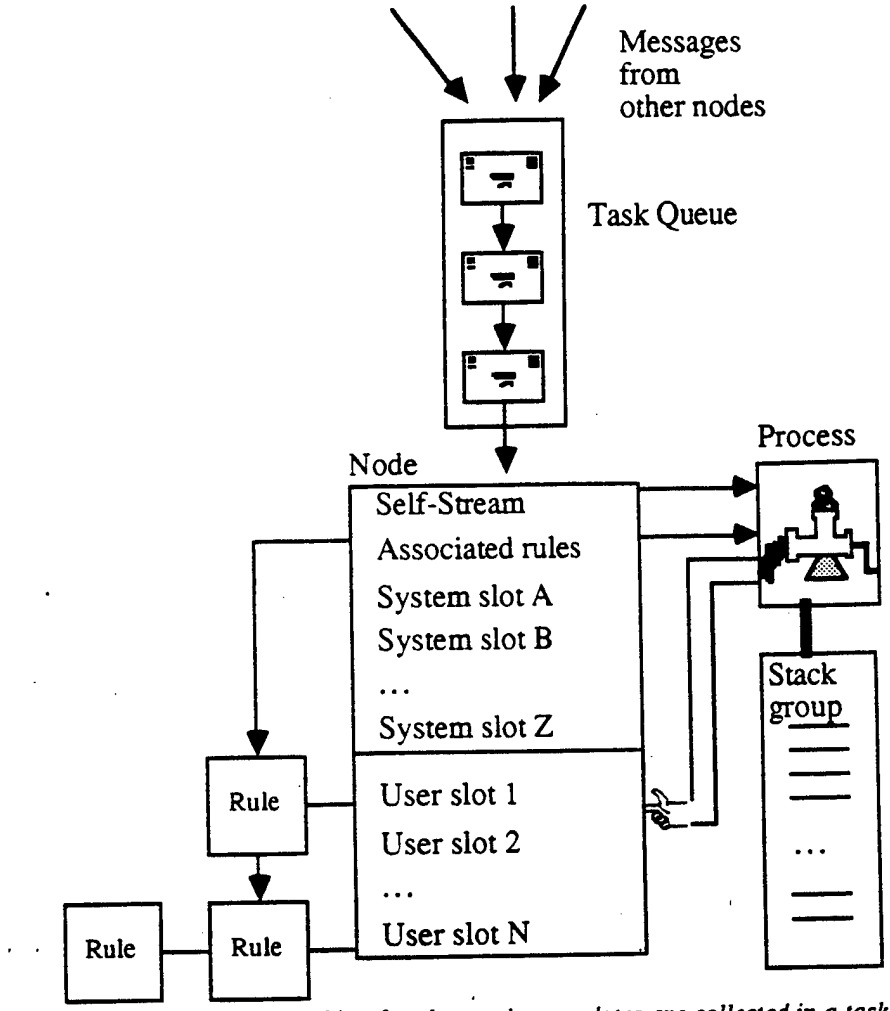


Fig. 4-5. Messages asking for slot reads or updates are collected in a task queue associated with the self-stream of the Poligon node. A process associated with the node reads tasks from the stream and executes them. Slot updates can cause the invocation of rules, which start up in the same process.

In place of a central scheduling mechanism, Poligon provides a daemon-driven mechanism for knowledge activation. We decided at the beginning that we could probably make the system work by triggering the knowledge in the system as daemons on updates to slots. The applications written using Poligon demonstrate that this in fact possible. Unfortunately, project resources did not allow us to test any other, different invocation strategies.[1] As shown in Figure 4-5, updates to nodes, as well as requests to read slot

[1]On a number of occasions, for instance, we were interested in allowing rules to be triggered by more than one slot. This was not implemented because the meaning of this deceptively simple goal is not at all obvious. What do you do when one slot is triggered but not another? Do you go into a partially triggered state and wait until the other slot is triggered? If you do, how long do you wait before you decide that the rule should not fire? All of these issues seemed too hard to tackle when we were investigating so many other new areas. Any new system that is to some extent like Poligon, however, would probably benefit from allowing rules to be triggered by patterns of slot events.

values, arrive in the task queue associated with the self-stream of a node. These are read by a process attached to the node, which loops continuously, looking for new things to do. When the process finishes its task, it informs the operating system on its processing element and is suspended, waiting to be reactivated when more work arrives. The operating system has the job of selecting a process from the set of processes with tasks on their self-streams and starting it up.

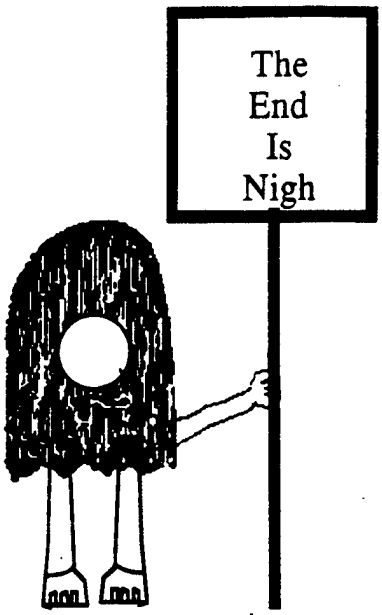
As mentioned above, we determined that using an architecture requiring numerous heavy-weight processes would be a major problem in any Poligon system that we could actually implement, and so we had to rethink the implementation model. We found that by taking advantage of the behavior of the CARE processors' operating system it was in fact possible to find a modified implementation model that would be significantly more efficient.

An important design decision in the CARE machine has been not to allow preemptive scheduling. This greatly improves the efficiency of the operating system and has some significant effects on the semantics of programs that run on a CARE machine. Because all processes communicate by sending messages to each others' self–streams, and because the operating system cannot preempt a process in a random part of the computation by the operating system, other than for fault exceptions, any given process can be sure that the node with which it is associated will not have been changed by another process since it was last activated. This means that in the time between the processing of tasks, the process for the node has unwound to the top level and has no state left on the stack. The process can be switched without the cost of a stack-group switch. This model of computation would, in itself, require the use of only one stack group on any processor, except when the system is intended to do useful work during fault handling. Nevertheless, the number of active stack groups would generally be small.

The problem with this model, which is in effect the Lamina programming model, is that it does not allow the general use of futures. This is because, in the absence of a really smart continuation passing compiler, it is not possible to construct a continuation for every possible point in the program where a future might be defutured, and so futures would not be allowed to be first-class citizens in the source language. They could be used only in very special ways at special times. This did not seem to be consistent with the programming model of Poligon, in which we wanted to preserve the abstraction of procedure-call semantics and a clear source language. For instance, we wanted always to be able to write the expression `«a» + «b»` at any point in the source code, whether or not the expressions `«a»` and `«b»` involved the defuturing of futures. Because in general a programmer simply could not know whether any given piece of data would be a future or would contain futures, we could not expect the user to write complex code to form the continuations for every such case.

What we did in Poligon, therefore, was to try to allow the semantics of generalized futures and yet still try to minimize the number of allocated stack groups. We had originally designed the system on the assumption that a "Poligon machine" would have some sort of hardware or microcode support for trapping access to futures on strict operators, so that the compiler would not have to insert special code for this case. Because of the difficulties of simulating this behavior and the lack of a real Poligon machine, we decided to use the compiler to try to minimize the amount of code needed to check for futures. Through the use of compile-time strictness analysis of the arguments on all functions called in a Poligon application and through the use of type declarations and type propagation, the Poligon compiler is able to deduce areas of code that could not possibly involve defuturing thereby eliminating any code that might have to check for this eventuality. This design had the beneficial property that defuturing would still be a lazy process, but it is still not as efficient as

a hardware implementation would be. Thus a Poligon process would block on a future only at the latest possible moment, allowing the maximum possible time for the future to become satisfied in the background. In fact, futures were often satisfied by the time they were touched, and so a process switch and the need for another stack group was avoided.



Waiting for a Future.

An ideal implementation of the Poligon computational model would be to start up a process to service a message running in a default stack group. This process has no initial state other than the message arguments and the instance variables in the Poligon node. This means that no special initialization or rebinding has to be done to start up the process. This need be only a procedure call. In most cases the processing of a message will not block on a future, so the stack group will unwind back to the top and a new process can be activated without significant cost. In the event that blocking on a future is necessary, a stack-group switch is then performed, swapping out the process and using a new stack group. Stack group switches are, therefore, done lazily.

Poligon's actual implementation of the equivalent behavior is not done in the same way, owing to the CARE simulator's design. CARE distinguishes between fully fledged processes that can be suspended and those that can only be restarted. It does not allow the user to decide halfway through a computation that a process is going to be suspendable. The cost of suspendable processes in CARE is quite a bit higher than that of the lightweight restartable processes. We therefore implemented a scheme whereby the process that actually reads the tasks from the self-stream of the Poligon node is lightweight and restartable one. On the basis of the arguments to the message it has been passed the process tries to prove to itself that a message can be handled without the need to suspend the process. It does so on the basis of the arguments in the message and information that the compiler deduced about the code fragments to be executed. The process is frequently able to deduce that the message can be handled without blocking and so it simply executes the task. If it cannot prove that the message can be handled without blocking it then has to make sure that the message is handled in a fully fledged process. This is done by acquiring a process from a resource of free, suspendable processes and then sending it the same message that was read from the task stream. The restartable process then suspends itself to wait until a reply comes back from the server process. It does this by suspending itself and waiting, not on the self-stream that it normally waits on, but on a stream that is private to these two

processes. Thus, the lightweight process that serves the Poligon node can be sure that nothing will be done to the node until the server process returns. This entire procedure is shown in Figure 4-6.

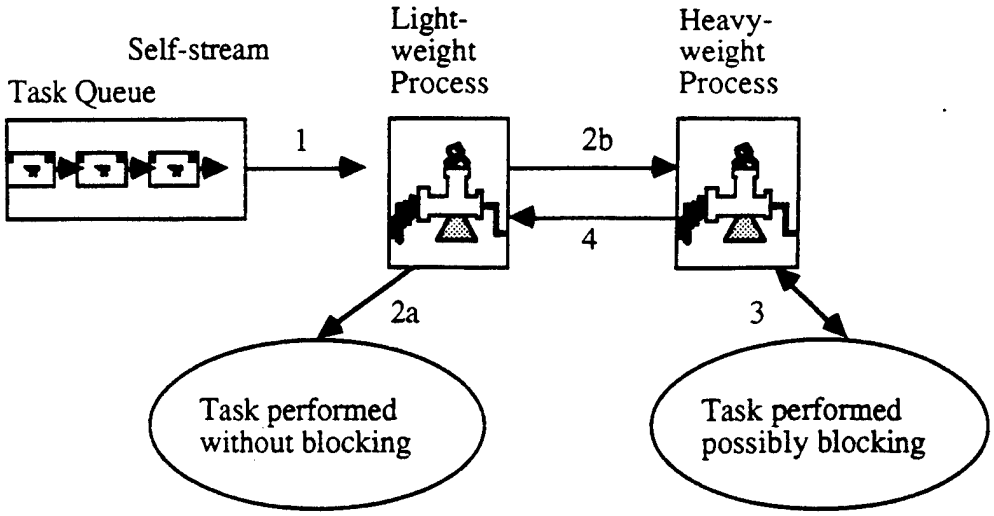


Fig. 4-6. The implementation of Poligon's process model. 1. A message arrives on the self-stream. 2a. If the message can be handled without blocking, it is processed immediately. 2b. If the message may possibly block, a heavyweight process is allocated and told to process the initial message. 3. The original message is processed, possibly suspending itself to wait for futures. 4. The server process replies to the node's process by a private stream.

This implementation model is considerably more expensive than a model that one would use in a production quality system. It involves the cost of trying to deduce whether a process can be handled without blocking, the cost of allocating the server process, the cost of sending the message to the server process, the cost of performing the stack-group switch to the server process and back again, and the cost of servicing the message that contains the reply from the server process. One would not implement such a model on a real machine in the field.

In retrospect, this design seemed to work reasonably well. Empirically the number of stack groups that were ever active was generally much lower than the number of Poligon nodes, and was generally a few times the number of processing elements in the system being used. This was not always the case, however. Occasionally a system would become very backed up because of real-time demands or pathological load-balance problems. As a result, a large number of processors had processes blocked, waiting for replies to messages sent to one processor that was too heavily loaded to service all the requests. It is clear that the model used by Poligon breaks down in such a case. Although it still gives the right answers eventually when all pending futures eventually receive the values they are waiting for, the model does not degrade as gracefully as one would like in instances of poor load balance.

4.4. Reading from Slots

> Dogberry: *Come hither, neighbour Seacoal. God hath blessed you with a good name: to be a well-favoured man is the gift of fortune; but to write and read comes by nature.*
>
> — Shakespeare, *Much Ado About Nothing.* act III scene III

The slot read operation in AGE, as mentioned above, was implemented as a function that used `Assoc` to find the matching slot being sought in the slot AList of the blackboard node. In Poligon we decided to use positional slots in order to achieve optimum performance.

In fact, we attempted a number of different implementations for Poligon's slots and the means of reading them. This was done as we learned more about the process of problem solving in parallel.

4.4.1. The First Implementation of Slots

The initial implementation of slots in Poligon nodes was simply as lists of values (see Figure 4-7). The user defined the slots that a node would possess in a set of class declarations, which were compiled to produce a suitable set of Flavors for the nodes on the blackboard. For instance, the user could say the following:

```
Class Aircraft :
    Slots :
        Wings
        Wheels
```

This would define a class called `Aircraft`, all of whose instances would have two user-defined slots, one called `Wings` and another called `Wheels`. Similar syntax was used in order to define metaclasses and to mix different superclasses together to implement more complex classes.

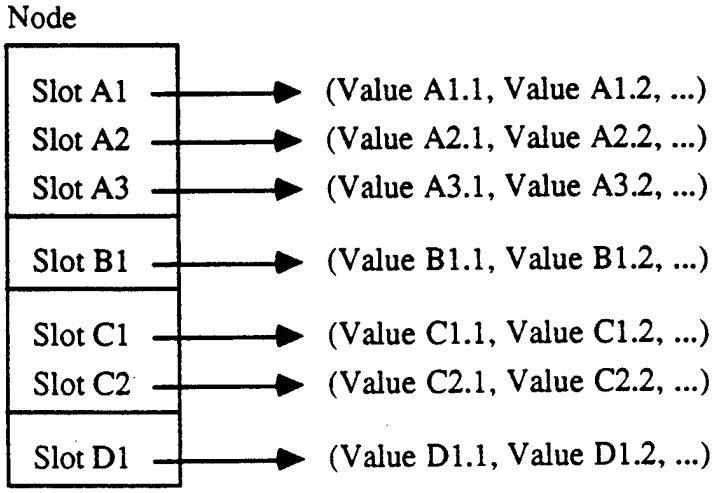


Fig. 4-7. The first implementation of slots for Poligon nodes was simply as value lists.

Operators in the Poligon language allow the user to read the values from a slot. For instance, `foo•wings` would read the first value from the `wings` slot of a node denoted by `foo`, and `foo⊕wheels` would deliver the list of all `wheels` associated with `foo`.

We quickly found that this was not sufficient. Even though the user could define operators to implement different functionality, because we wanted to support real-time systems in Poligon, we needed some sort of support for timestamping. Data would arrive out of order, and we needed some way ensure that the system would not get confused by the value lists of slots not being in strict temporal sequence.

4.4.2. The Second Implementation of Slots

The next implementation involved a form of automatic time-stamp propagation. Each element in the value list of the slot was encapsulated within a data structure that also contained a timestamp for that value. This is shown in Figure 4-8. These timestamps were set when the data entered the system and were propagated throughout the blackboard during problem solving. When the user evaluated an expression, for example, `«a» + «b»`, and stored the result in a slot, the new value would be timestamped with the time at which the actual computation of the expression `«a» + «b»` finished. Thus, the system could always associate a time with every slot value. To use these timestamps, we introduced a number of new operators. Whereas previously an operator such as "•" would simply read the first value from the value list the new operator "•↑" would sort all the values in the slot if they needed to be sorted and would then return the most recent value according to the timestamps associated with the values in the slot.

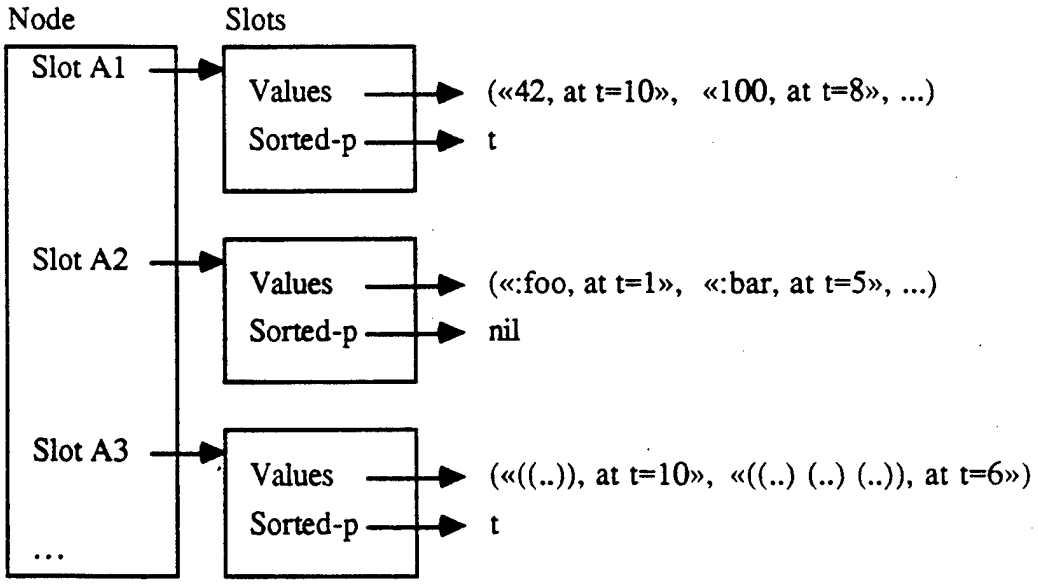


Fig. 4-8. The second implementation of slots for Poligon nodes caused each slot to contain a slot object that had a list of values and a flag that indicated whether or not the values were sorted.

We also found early on that the user often read a number of slots from the same node throughout the body of a rule. This was problematic, because an expression such as `foo•wings = foo•wings` might not be true if the value of the `wings` slot was modified by the time the second read was performed. We needed a way to capture a consistent view of blackboard nodes.

This was addressed by the implementation of a block read construct. Since the program was already sending a message to ask for the value of one slot, the marginal cost of asking for a number of slots at the same time was reasonably low. In Poligon it therefore became possible to write expressions such as `foo&•wings&•wheels` in order to read the values of the `wings` and `wheels` slot within the same critical section. This formalism proved to be very useful and was used in all later designs of slot-reading behavior.

The second slot implementation mechanism worked reasonably well but we found that it was rather expensive and suffered from some major flaws. It was frequently the case that the user's data already had timestamps of its own, which were often at variance with the timestamps that the system had imposed. In addition the user often preferred that the values be sorted on a basis other than the time in the timestamps. This led to the inclusion of an operator that allowed the user to force any value into the system timestamp slot whether it represented a time or not. A secondary flaw was that the user often wanted to index the data in a slot. For instance, when a rule was triggered and was interested in something that happened at time `t`, it would frequently want to know about other things that happened at time `t`, or perhaps about things that happened at `t-1`. Getting data on the basis of such an association was not simple to do using the simple block read mechanism described above, since it involved getting all of the values in the slot, possibly from a remote location, and then searching them for the data required.

4.4.3. The Final Implementation of Slots

In developing our third approach we decided that operations like sorting and indexing were fundamentally important to the ease of programming a Poligon application, but that the system should not impose any unreasonable restrictions on the things that could be sorted or the things that could be indexed. It was therefore decided that these operations should be user defined, but also that Poligon should provide the user with some sophisticated and abstract mechanisms for the expression of his program.

Each slot was implemented as an object. In fact, a different Flavor was created for each slot of each class in the system. This allowed the user to specify behavior in a highly focused, per-slot manner, that is, each slot could have specialized behavior associated with it. For example, the user could specify that the values of a slot were to be sorted by a particular predicate, or that they were to be accessed as mappings from indices to values, using a particular index function, or both sorted and indexed. Thus the user could write the following code:

```
Class Aircraft :
    Fields :
        Wings :
            IndexedBy : 'Wing-Side
        Wheels :
            SortedBy : '<
            KeyedBy : 'Tyre-Size
```

As before, all instances of the class `Aircraft` will have two slots, but in this case it is possible to find a wing in the slot `Wings` by looking up which side the wing is on. It is also possible to view the `Wheels` slot as a sorted list of wheels, which is sorted according to the tyre size of those wheels (see Figure 4-9). The "•" operator mentioned earlier was modified to allow the specification of an index, so the user could get the left wing of a node

`foo` with the expression `foo•wings At :Left`, and could get the smallest wheel with just the expression `foo•wheels`.

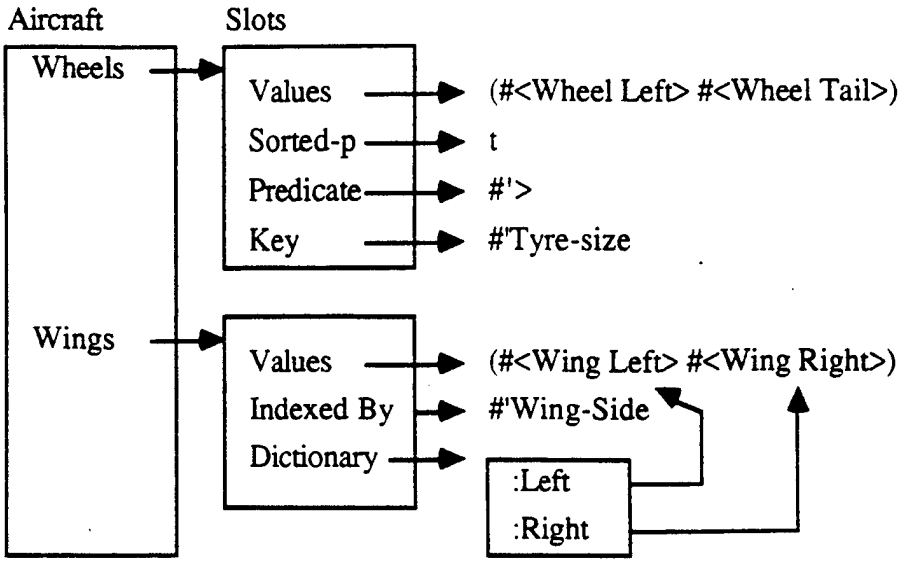


Fig. 4-9. The final implementation of slots for Poligon nodes made each slot point to a slot object, which was of a specialized type unique to that slot. Wings can be seen to have dictionary-like behavior, and Wheels are sorted according to the size of their tyres.

This implementation was more expensive than is strictly necessary. Suitable compilation could expand the state associated with the slot objects into the node itself and could reduce an access simply to an array offset. This would not be in any way incompatible with the compilation of slot accesses into fixed-position array accesses. Similarly, the methods associated with the operations supported by each of these slot types could be fully compiled. In fact, there is no particular need to use methods, in the Flavors sense, since the combined methods for each slot are known at compile-time and no complex method combination is required. It seems likely, therefore, that although this is a considerably more expensive implementation than the original AGE use of a value list, it could be made reasonably efficient and, more important, is considerably more useful in a parallel environment.

4.5. Writing to Slots

The implementation of writing to slots in many ways mirrored the implementation of reading them. This is by no means surprising. The major difference between the evolution of the slot write mechanism as opposed to that of reading slots was that the the ability to write multiple slots at the same time was in Poligon from the start. We did this because the same was also the case in other blackboard systems like AGE.

4.5.1. The First Implementation of Slot Updates

As mentioned previously, Poligon's slots originally had no structure to speak of; there was nothing particularly special about the slot-updating process. AGE supported a pair of frequently used slot update procedures called $Modify and $Supersede. $Modify tacked a new element onto the front of the value list of the slot being side-effected, and $Supersede had the effect of overwrote all the elements in the value list. In effect, $Modify was an optimization for a frequently used case, and $Supersede was an implementation of the general

case. It was possible to read all the values of the slot, perform an arbitrary transformation on them, and then write out a new value list at the end at the end of the rule's execution.

We did not think that this would be sufficient in Poligon, so we implemented the functionality of $Modify and $Supersede as slot update operators, which could be user defined.

The result was that these operators, at least with their AGE semantics, were almost useless. For the reasons already described, things arrived in the slots out of sequence. The Poligon equivalent of the $Modify operation, which mirrored the "•" operator for slot reads, would not really do the "right thing" because one would prefer that the value be put into a more specific place than just the front of the value list. This was due to the fact that the value lists were implicitly time ordered. The Poligon equivalent of the $Supersede operator proved to be almost useless on account of a hidden critical-section assumption in the AGE model. It was not possible to read all the values of a slot and then write a new list back out again because there was no guarantee that the slot had not been side-effected between the time the slot was read and the time it was written. Writing out a new value list would then destroy any new results that were put in by other rules during the computation.[1]

4.5.2. The Second Implementation of Slot Updates

When the slot read mechanism was changed to support timestamping (see Section 4.4.2], we then had a way to avoid the problems we had had with slot updates. Because so much more data was now available to allow the programmer to perform more sophisticated slot updates, there was an explosion in the number of Poligon's slot update operators, which would, for example, *remove an element if it was already present* or *add a new element unless it was already present and was not Nil.*

This added considerable complexity to the programming task. The user had to know about the semantics of the operators that the program would use to *read* a slot in order to pick the correct operator that would *write* that slot. A better abstraction was required. We also noticed that many of our slot update operators seemed to be explicitly fault tolerant. They were all trying to do the "right thing" in the event that the shape of the data in the slot was not quite what was expected. This proved to be an important observation, because it allowed us to develop a much more satisfactory programming methodology and then to build our slot update mechanism around it.

4.5.3. The Final Implementation of Slot Updates

> *All science is either physics or stamp collecting.*
>
> — Ernest Rutherford.

Our final implementation of the slot update mechanism used the methodological idea of "smart" slots. As noted earlier, slots had already been modified by the inclusion of some general mechanisms so that they could be read in ways that were highly application specific. New mechanisms were implemented to support much more focused slot update behavior. The user could now express ideas like *remove this element if it is still there*, and *add this element if it is a new one* in a manner that was both declarative and abstracted out into the class declarations. For instance:

---

[1]This assumes that the whole bodies of rules are not executed within critical sections on the nodes that trigger them. This is discussed at length in Section 4.7.

```
Class Aircraft :
    Fields :
        Wings :
            RemoveIf : 'Still-Present
        Wheels :
            InsertIf : 'Not-Present
```

In this example the user-defined functions `Still-Present` and `Not-Present` will be called whenever the program attempts to put new values into these slots or to remove them. In fact, Poligon's default behavior is to do reasonable things in such cases, but this serves as a simple example. The crucial point is that the slots of nodes are now expected to be responsible for their own upkeep. They are intended to have evaluation functions that express the intention or purpose of the slot and thus are capable of assessing any update that is requested and deciding whether to perform it, ignore it, or perform some different update. This results in a sort of local hill-climbing behavior, which allows some Poligon applications to iterate toward a globally reasonable solution. These functions are defined and stored in the same per-slot manner that the sorting and indexing functions are for read operations. They are just extra instance variables in the slot objects that represent the slots.

The functions that the user can specify can be arbitrarily complex. This means that the user has the ability to put arbitrarily expensive code into the slot update critical section. This will clearly lock the node for a long time, but it is better to do it slowly and right than quickly and wrong.

4.5.4. Test-and-Set

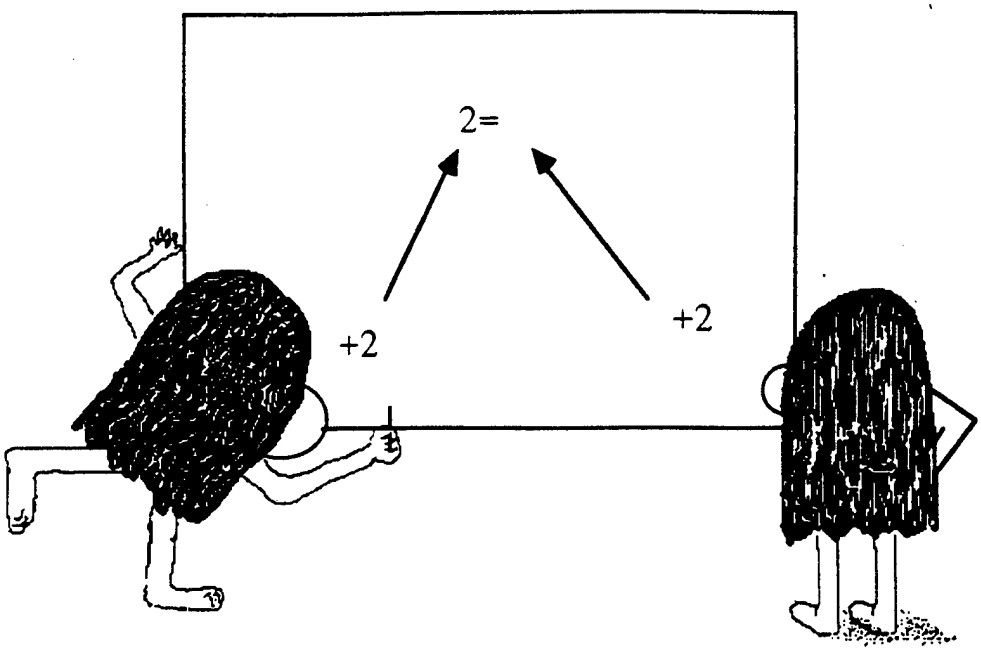
There is one more point to discuss regarding slot reads and writes: we found it necessary to implement a test-and-set operation. Although Poligon nodes are now responsible for keeping themselves reasonably coherent, that does not help us if we really need to perform some sort of atomic read/write operation, such as one might want when implementing locks or performing accuracy-critical database operations. Without some sort of atomic test-and-set operation, one would not want to use Poligon to implement a bank transaction system.

We therefore implemented such a test and set operation. The user can now express ideas such as the following:

```
foo•wings
    Unless : foo•wings = 2
    Updated Fields :
        wings ← 2
```

In this case, if the node `foo` has two wings then this value is returned; if it doesn't, then `foo` is given two wings and the original number of wings is returned. The test-and-set operation has been used only twice in Poligon applications, but in these cases there didn't seem to be a way of implementing the program without it.

4.6. Creating Instances



Eagar finds that unmanaged instance creation can lead to the wrong answer in a concurrent problem-solving system.

The creation of instances is something that serial systems typically do not handle in a particularly sophisticated manner. The reason for this is that users generally write their knowledge sources so that they are large enough that they know that they are doing the right thing when they create a node. This is, in effect, using large critical sections in order to guarantee that the blackboard is consistent throughout a node creation operation. If knowledge sources are not large, however, especially in processing, it is quite probable that they will create multiple nodes representing the same real-world object. This happens frequently in a parallel blackboard system, and so some mechanism is needed to deal with it.

```
New Instance of Aircraft
  Unless :
     Associate(Id-Number, Aircraft⊕Cache,
               :Return #'Second)
  Updated Class Fields :
     Cache ← List(Id-Number, The-Created-Node)
  Initialization :
     Wings ← 2
     Wheels ← 3
```

Fig. 4-10. Poligon language source code to create a new instance. If there is an entry in the cache slot of the class node called Aircraft, which is a list of the form ((id <node>) (id <node>)), then the node is returned. If there is no such entry, a new node is created. The new node has its wings and wheels initialized, and the class node's cache slot is updated so that it has an entry for the new node. The node that has just been created is referred to by the name The-Created-Node.

Two options were considered. Either one could manage the creation process so that only the needed nodes are created or one could add extra knowledge so that the system could reason about the presence of a number of nodes representing the same real object. The latter would, in the general case, require application-specific knowledge in order to achieve this goal, whereas the former could be implemented in a manner that provides a domain independent means of handling node creation. We picked the strategy of managing node creation, knowing that the price would be serialization.

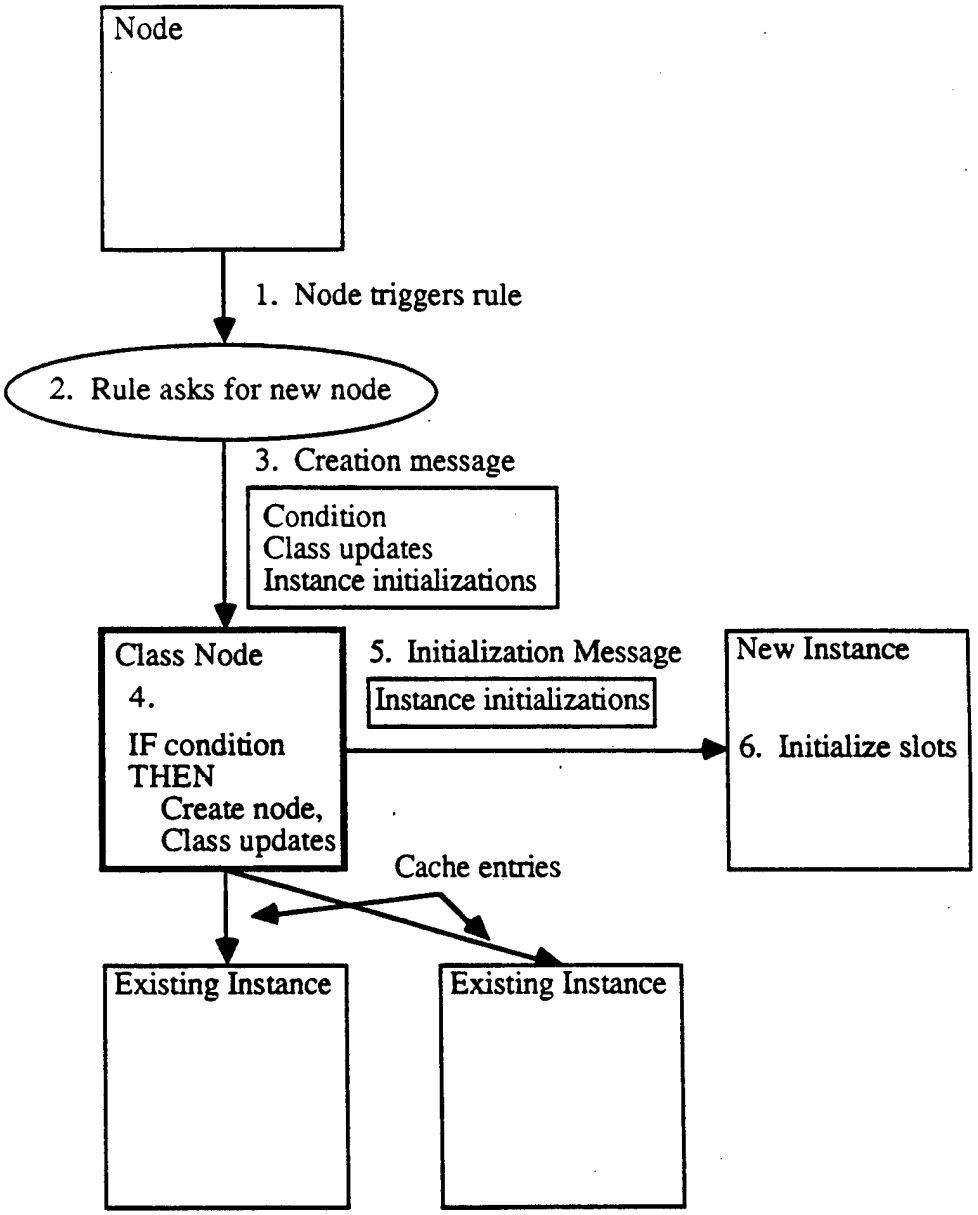


Fig. 4-11. Managed node creation in Poligon. 1. An update to a node triggers a rule. 2. The rule that fires decides that a new instance must be created. 3. A message containing the condition, class update, and initialization closures is sent to the class node for the class to be created. 4. If the condition allows it, the new node is created, the initialization closure is evaluated and passed to the new instance (5), and any class updates are performed. 6. When the initialization closure arrives at the new node, the new node's slots are initialized.

After some experimentation, we developed a fairly complex but general node creation and initialization mechanism for Poligon. We needed to be able to allow the conditional creation of nodes because, in the general case, a node that represents the thing that we're interested in may already have been created. We needed to be able to construct mappings from identifiers to nodes. These mappings have to allow us to determine whether we already have a node to represent a given real-world object. Finally, we needed to be able to initialize the new node appropriately and make sure that all the right things are executed atomically.

An example piece of Poligon code to create a node is shown in Figure 4-10. The way that this code works is explained below and shown in Figure 4-11.

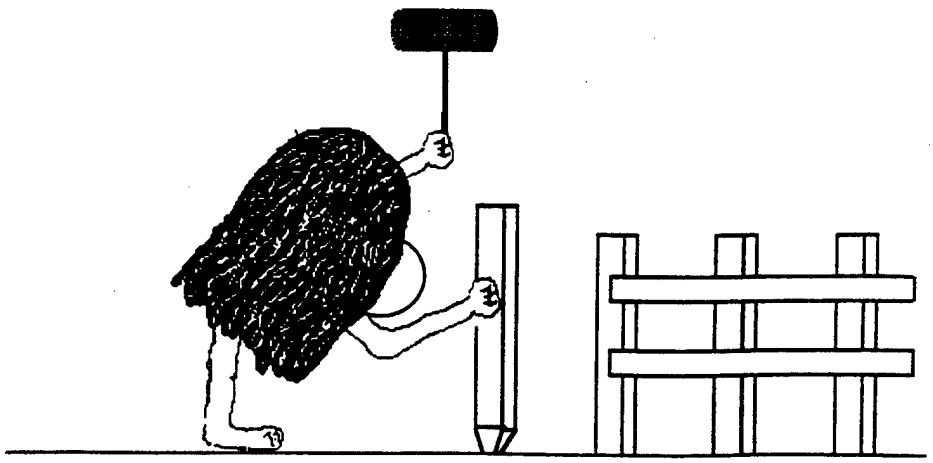
- First is sends a message to the class node that represents the class of node to be created. This message tells the class node that a new node is to be created or an existing node is to be returned. The message contains three (possibly null) closures as its arguments: a condition closure, a set of class node updates, and an initialization closure for the node to be created. These closures close over the environment of the rule execution so that the program can make reference to expressions in the context of the rule as well as to expressions in the context of the class node or the node being created. Any expressions that require references to be made to anything other than the class node or the node to be created are evaluated before the message is sent. This allows the class and the new node to execute their code without blocking. The expression that asked for the creation of the node returns immediately with a future to the new node. A rule, therefore, need not block in order to create a new node.

- When the message is processed by the class node, the condition part, if supplied, is executed. The condition is executed on the class node because, in the general case, the expressions that make up the condition will want to make reference to caches that are held on the class node.

- If the condition evaluates to `Nil`, a new node is created and entered into the class's list of instances (this process seems reversed, but it actually works.) If the expression is not `Nil` it is taken to denote a preexisting node that represents the solution-space component (node) that we really want.

- If a new node is created, the initialization closure is evaluated. This is a closure that is executed in the context of the class node so that reference can be made to class slots such as the identifier cache. The result of the evaluation of this closure is another closure that is sent to the new node. It is this second closure that actually performs the initialization of the new node's slots. By this point the closure will have picked up all the context it needs from the originating node and the class node. Any rules associated with the slots being initialized will fire for the new node.

- The newly created node is seen by the class node as a future. The class update closure is now invoked with this new node visible. The update closure is therefore able to add the new node to the id cache, even though that node may not yet have been created.



Eagar closes over his environment.

This instance creation mechanism is very general and works reliably, but a number of possibly unnecessary overheads are associated with it. For instance, it may not be necessary to manage the creation of the node. If the node is being created as the result of an unique piece of signal data, the programmer knows that only one node will ever represent this object. Serializing through the class node is unnecessary in this case. In practice, we found that creating nodes to represent low-level objects to represent signal data tended to overload the class node if the instance creation mechanism outlined above was used. Thus, we implemented an optimized form of node creation to allow for this special case. This is shown in Figure 4-12 and works as follows:

- It create an instance directly without reference to the class node and immediately returns with a future to the created node.
- It initializes the node only from the context of the invoking rule, not from the class node.
- It sends a message to the class node telling it that a new node has been created.

Here node creation is unmanaged but faster, but there is another, sometimes undesirable consequence of this operation. Poligon supports operations that can be performed on all instances of a class. Because node creation happens in parallel with the notification of the new node's class, it is possible for the message that notifies the class to be delayed and thus for other pieces of knowledge to execute under the assumption that they are referring to all the instances of a class, but actually they are missing the newly created one. This generally doesn't seem to be much of a problem in Poligon applications, since one usually has to use the full, managed node creation mechanism anyway, in which case the problem doesn't occur, or one's program is generally written so as not to be brittle to this sort of inconsistency.

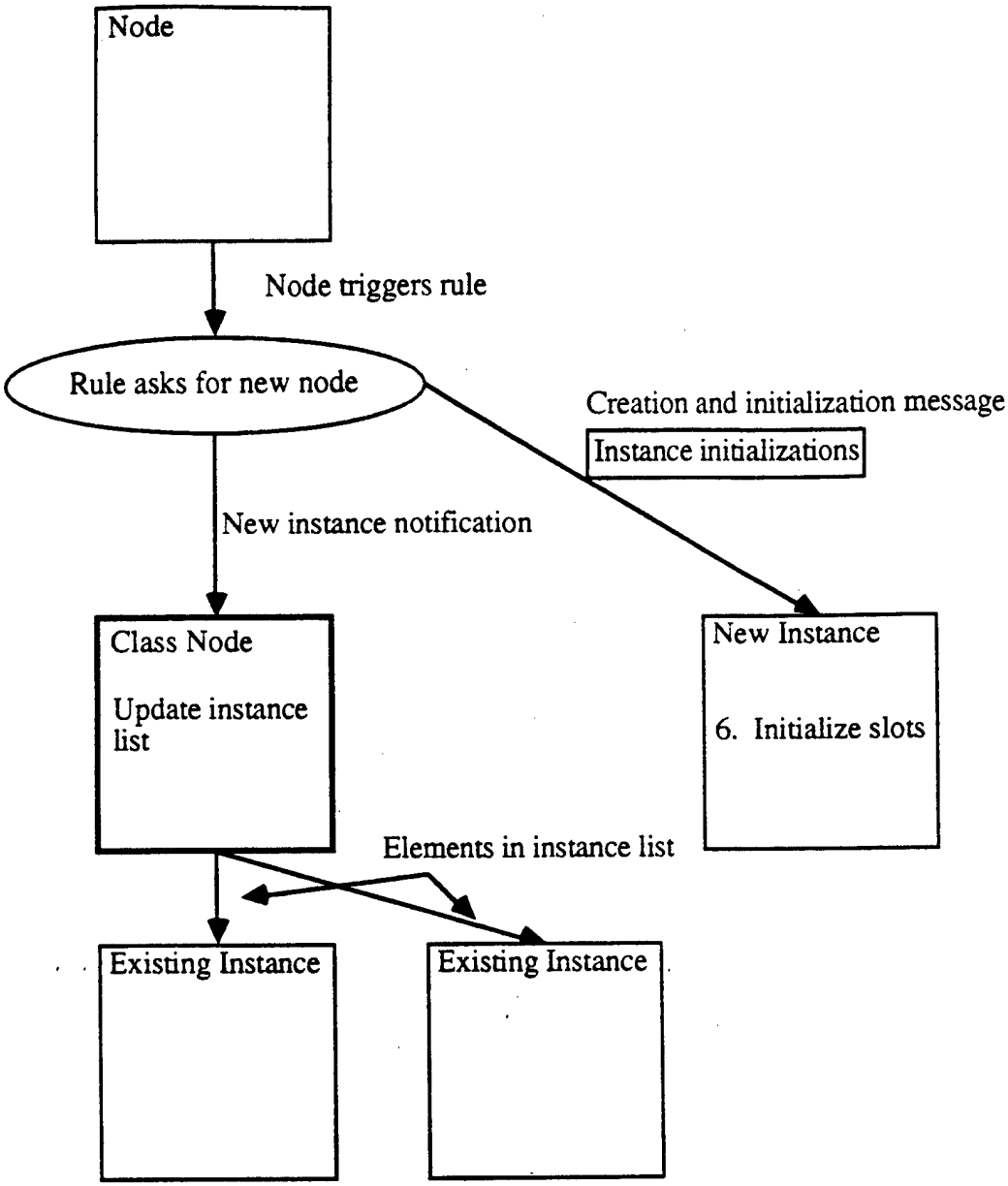


Fig. 4-12. Optimized node creation. A rule triggered from some node decides that a new instance should be created. The rule invocation creates the instance directly. The class node is notified about the new node by having a future to the new node sent to it and the class node can then add the new node to its instance list.

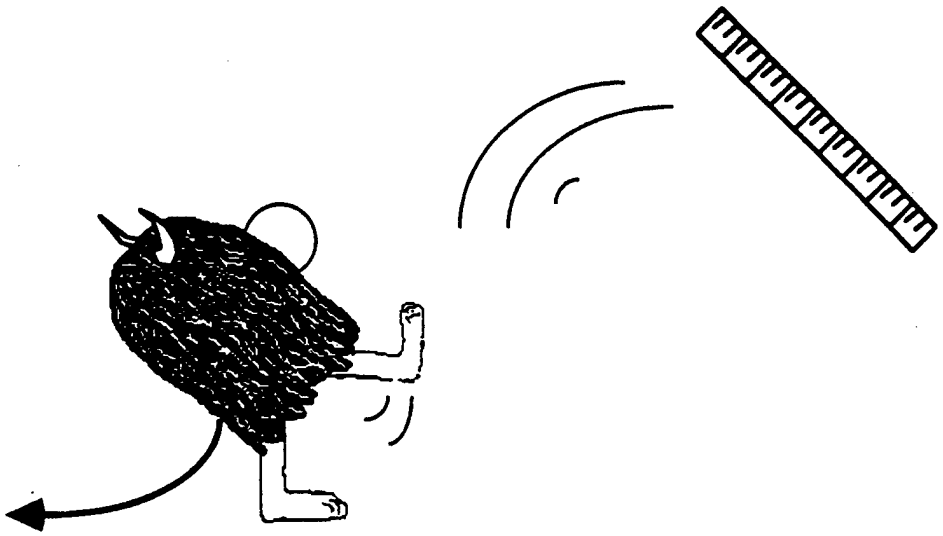
4.7. Rule Invocation and the Context of Rule Invocation

Many assumptions about the granularity of the Poligon system were made throughout the development of the system. Perhaps the most significant of these was the decision to allow concurrent rule execution on blackboard nodes. The cost of process creation and/or switching is always going to be significant in the design of a system like Poligon; so the decision that concurrent rule execution should be allowed on each blackboard node necessarily carried with it the assumption that nodes would, in general, have a lot of applicable knowledge at any given time. A second assumption was that the cost of the computation

performed by that knowledge would be significantly greater than the cost of rule invocation.

In this section we discuss the implementation of the rule invocation mechanism in Poligon and other related topics.

4.7.1. The Triggering of Rules



A daemon triggers a rule.

As mentioned above, rules are triggered as daemons on the slots of nodes. The slot that triggers a rule is defined by the programmer and is fixed at compile-time.[1] It is not possible, however, simply to fire the rules as soon as the update that would trigger a rule is made. This is because Poligon supports the ability to update a number of slots "simultaneously." To allow rules to be activated before all relevant slot updates had been performed would open the door to very counter-intuitive behavior. Poligon, therefore, collects the significant events on slots as the slots are being updated, and when all of the updates have happened, activates the interested rules.

The evolution of the slot update mechanism was discussed in Section 4.5. A consequence of this implementation is that the system always knows what changes to a given slot were made when it was updated. For instance, if a new wheel were to be added to an aircraft, then the node being updated would remember the node that caused the update and the slot that was updated, the new wheel — or, actually, the set of new wheels, which in this case is a singlton set.[2]

Once this information has been gathered for the updated slots, the node is free to trigger the associated rules. For this the node must create contexts.

---

[1]This is not the case for expectations, which are described in Section 4.7.4.

[2]A deficiency of the implementation in Poligon is that there is no way for a rule to recognize whether the values that it is passed, which tell it what caused the rule to fire, are values that were added to or removed from the slot. The rule has to work this out for itself. Clearly this would be a small thing to fix but is worth noting here, since we have found programs that wanted to trigger rules both as a result of inserting an element in a slot and as a result of removing values.

4.7.2. Contexts and Pseudo-Contexts

In AGE, the context in which a rule is executed is that of the knowledge source and the knowledge source bindings. The knowledge source knows only the token that triggered it and the node that caused the triggering.

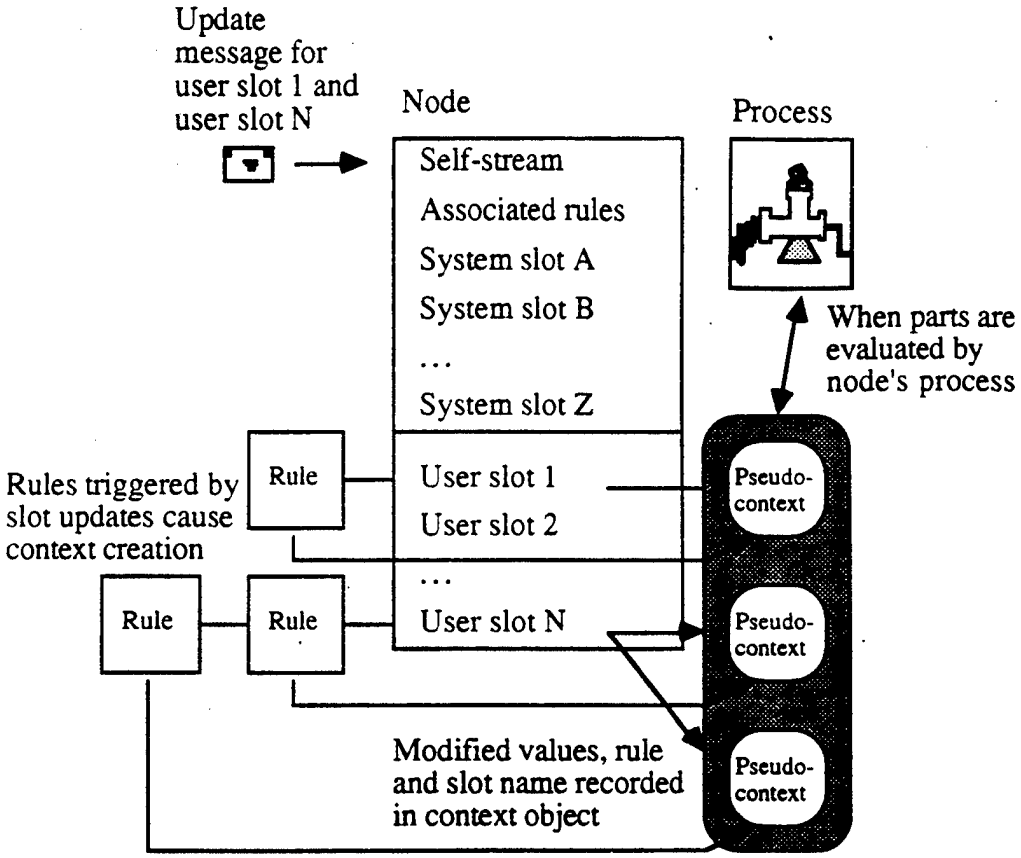


Fig. 4-13. A node receives an update message. The updates, when processed, cause the triggering of the rules watching the slots that were updated. When a rule is triggered, a pseudo-context is created and the When part of the rule is evaluated locally.

In Poligon we decided early on that knowledge sources were too coarse grained to give us the performance we wanted. Consequently, we wanted at the very least to execute rules in parallel; we wanted to compile away the knowledge sources, preventing them from being the scheduling units that they are in serial blackboard systems, and to incorporate into the rule any state that the knowledge source might have had. Poligon has functionality equivalent to knowledge source bindings called *definitions*, which are discussed in Section 4.8. Moreover, a number of existing blackboard systems have a more substantial amount of context when their knowledge sources are activated than was available in AGE. BB1's [Hayes-Roth 85] knowledge source activation records (KSARs) are an example of this. If we were to run our rules in parallel, we knew that we needed some representation of the rule's evaluation context, and that we had to associate a process with each rule in order to execute it. In addition, because Poligon runs on a distributed memory system, each element of which is effectively a uniprocessor, the activations of rules would, in the general case, be running on different processors from that of the node, which caused the activation of the rules. Each rule therefore runs in a different address space from that of the triggering

node (see Figure 4-13). The objects that represent the activation of a rule in Poligon are called *contexts*.

Contexts are Flavors instances that contain the following information:

- The triggering node
- The rule being triggered
- The values of the slot that caused the rule to trigger
- The definitions for the rule being executed

Recognizing that the cost of a process switch to a context would be significant and that the reading of values from the focus node would be expensive for a context on a remote processor, we wanted some means of filtering out rule activations before they became too expensive. For this we used a cheap test called the *When* part of a rule. Poligon knowledge sources have no preconditions, since they are compiled away, but rules needed a cheap means of determining whether they were applicable or not. The condition part of a Poligon rule is therefore split into two distinct components: the When part and the If part. The When part is executed by the node for which the rule is triggered. This means that no process switch is performed to evaluate the When part and that slot reads to the node can be fast. The If part of the rule is executed only as long as the When part succeeds and is executed in a different process by a context object. It is therefore useful for the rule to do as much cheap filtering as possible during the When part and to perform enough reads to slots on the focus node in order to prevent the context from having to read from the node again if that is possible.

If we were to allow the node to read values from other nodes during the When part, the node would have to be able to block until the results came back. This seemed undesirable, because it could cause areas of the evolving solution to lock up for unpredictable periods of time. We thus decided not to allow the When parts of rules — or any parts of the user's program that are executed on the nodes themselves — to make remote references. These are only allowed during the If or Action parts of rules, which are executed by contexts.

In order to be able to evaluate the When part of the rule, the node must create a context for the When part's execution. We wanted to avoid the cost of process switching during the When part, so the node creates an object called a *pseudo-context*. A pseudo-context is just like a context, only it executes within whatever process invokes it. As a result, any state developed during the evaluation of the rule's When part is recorded in the pseudo-context. If the When part evaluates to true a context is invoked and is passed the state in the pseudo-context as part of its initialization message.

The reason pseudo-context objects are necessary is that the When parts of rules can contain references to definitions, which are described in Section 4.8. In retrospect, we can see a justification for having yet another sort of precondition, one that does not allow the use of definitions. The creation of the pseudo-context for the rule, although much cheaper than a process switch to a context, still carries a significant cost. If we could do some prefiltering without this cost, we could expect better performance. Substantial optimization of the pseudo-context creation mechanism could be made, but avoiding this altogether, if possible, seems worthwhile. A smarter compiler might, perhaps, have done some flow analysis on the source program and created the contexts lazily. This would have required a substantial reimplementation of the rule activation mechanism in Poligon and so was not im-

plemented, but it could well be a useful strategy for any future system built along these lines.

4.7.3. Rule Execution After the When Part is Evaluated

When the When part of a rule is evaluated and is true, the node that started up the rule must invoke a context to process the rest of the rule. Normal rules come in two forms in Poligon: If–Then–(Else), or If–Then–Case–Else. It was found early on that blackboard systems commonly have sets of rules whose form is of the following type:

```
Rule 1:
    If «some condition»
    Then «some action»

Rule 2:
    If Not[«some condition»]
    Then «some other action»
```

This is typically not too much of a problem in serial systems because the cost of evaluating such a pair of rules is often reasonably small, but the constraints of a parallel or high-performance system make it desirable to avoid unnecessary rule invocation as much as possible. It is not the cost of the user code that one wants to avoid, since the code will be evaluated nevertheless; it is the evaluation of the system code that creates the contexts for rule invocations and that starts up the rule. In the previous example, it is clear that the two rules are mutually exclusive. They could, therefore, be rewritten in the following form:

```
Rule 1:
    If «some condition»
    Then «some action»
    Else «some other action»
```

Poligon supports just such a rule representation and, in fact, generalizes it so that the following rule is possible:

```
Rule Watch-for-changes-in-wheels :
  Class : Aircraft
  Slot  : Wheels
  Condition Part :
    When : The-Wheels > 0
    If : The-Aircraft•is-airborne
    Select :
        If The-Wheels = 3
        Then :Land-ok
        Else :Belly-land
        EndIf
  Action Part :
    :Land-ok :
      «we have enough wheels to land on the
       undercarriage»
    :Belly-land :
      «do whatever you must to land on your belly»
  Otherwise Part :
      «we haven't taken off yet so the change must be
       due to the maintenance crew»
```

This trivial rule takes advantage of the mutually exclusive execution of its action parts. For any given change in the number of wheels on the plane the rule will be invoked only once. If we had a number of rules that watched for this change, two rules might start up because the plane did not have enough wheels, but by the time one of them actually came to look at the plane and do something with it, the other rule might have caused more wheels to be added, thus confusing the first rule. The form of rule shown above has proved to be powerful, useful, and efficient.

The Poligon compiler causes the rules that the user expresses to be split up into a large number of functions and methods. The objects representing the rules in the system have a number of slots that refer to the code to execute the different parts of the rules. A very simple piece of code run by the context object looks at the rule object that represents the rule that it is to fire and invokes the relevant parts of the rule as appropriate. This is shown in Figure 4-14.

In designing the Poligon language, we did not want the user to have to worry about picking the parts of the program that would run in parallel and those parts that would run serially. To this effect we designed the language so that the user expressed those parts to be executed serially rather than those to be executed in parallel. The system was then at liberty to execute any other code fragments in parallel if this seemed appropriate. Clearly we wanted to execute as much as possible in parallel, wherever it would be beneficial. Our intention, therefore, was to execute components of the action parts of the rules concurrently and without any synchronization.

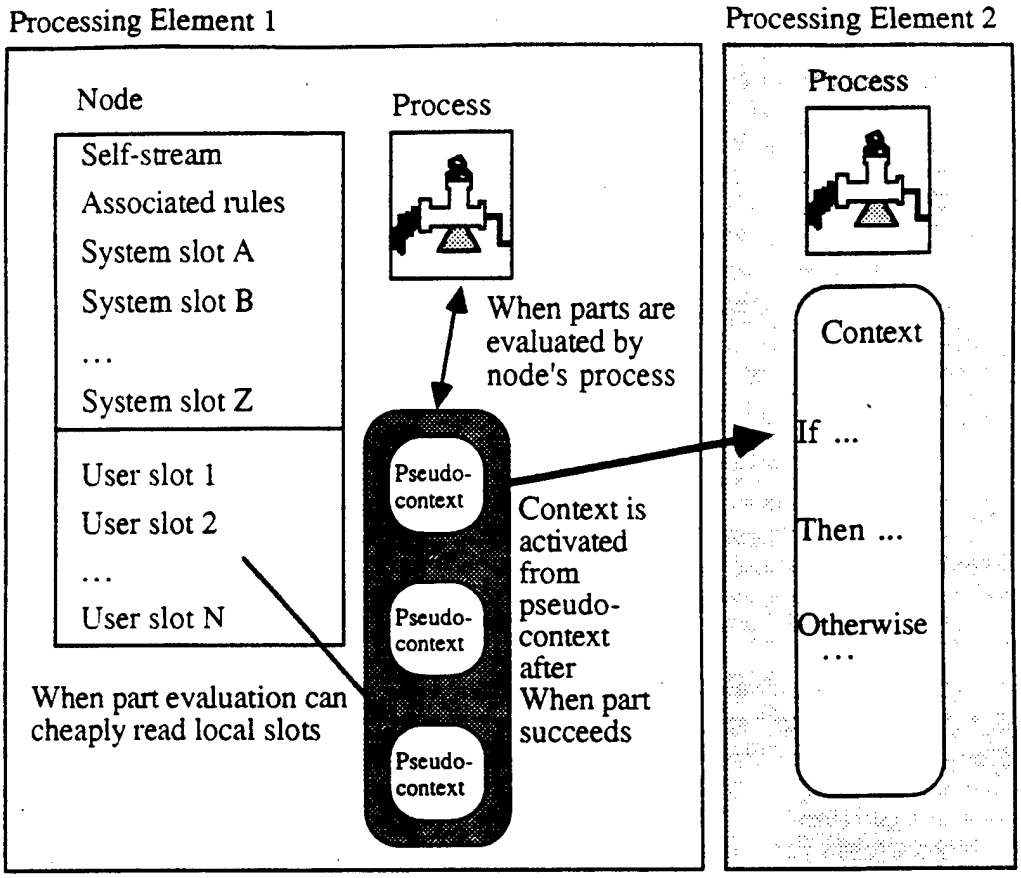


Fig. 4-14. When the When part of a rule evaluates to true, *a context object is activated, possibly on another processing element, to evaluate the rest of the rule. Cheap access to local slots can be made during the When part's execution. Slots can be read from the focus node during the evaluation of the rest of the rule but this is discouraged.*

The following fragment of Poligon code expresses updates being made to two different nodes:

```
Action Part :
  Changes :
    Change Type : Update
      Updated Node : «aircraft 1»
      Updated Fields :
        wheels ← «a new wheel»
        wings ← «left canard», «right canard»

  Changes :
    Change Type : Update
      Updated Node : «aircraft 2»
      Updated Fields :
        wheels ← «tail wheel»
        wings ← «new left wing»
```

We wanted the execution of these two updates not to be held up by each other. What we did was to make the Poligon compiler extract the references to any definitions (see Section 4.8) from the expressions in the action parts and evaluate them whenever possible before

entering the actual expressions that perform the updates. In order to perform the updates, the Poligon compiler tries to deduce the best place in which to evaluate the change component. In the cases above, each change will be evaluated on the respective aircraft nodes.[1] A message is sent to the node to do the computation, specifying a method that will be used to make the update and containing a newly CONSed pseudo-context that contains the definitions that are to be used in the evaluation of the action part. The use of a pseudo-context in this case allows a regular mechanism for the evaluation of definitions irrespective of where or when they are evaluated. The compiler has already computed which definitions will be needed in the execution of the action-part clause so only these are sent over. Figure 4-15 illustrates this sort of slot update operation.[2]

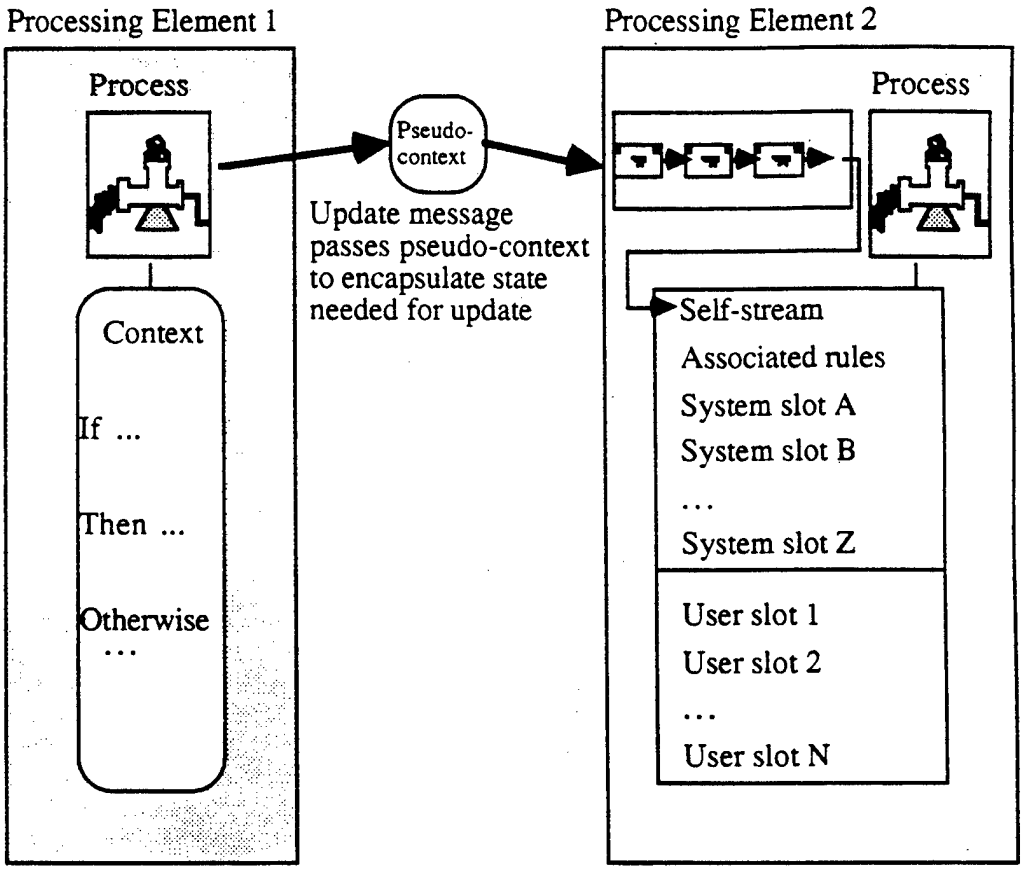


Fig. 4-15. When an update is required, a pseudo-context representing the required state from the current context is passed along with the update message to the node to be updated. The update is performed in the environment of the pseudo-context. Copying the state in the context allows concurrent execution of action parts without contention for the context in which the rule is executed.

---

[1]There is an assumption here that the code to be executed on these remote nodes will be reasonably cheap. It is left to the user to make sure that expensive expressions are factored out as definitions that are evaluated before the update message is sent.

[2]This whole design strategy may have been flawed. The creation of pseudo-contexts for the definitions passed to the nodes to perform the action parts is costly. It would probably be better to compile the action part into methods that accepted the definitions as arguments. This, however, would have the consequence that all definitions would definitely have to be evaluated before any part of the action could be executed. As a result, the lazy semantics of the definitions would be lost if any definition references were made in expressions with conditional clauses. This strategy clearly requires more thought.

When the message for the update arrives at the node to be modified, it executes the necessary code to perform the update, extracting any definitions that it may need from the pseudo-context that it has been passed. As mentioned earlier, the compiler has already determined which definitions will be needed in order to perform the update. At run-time the context that evaluates the rule knows which of these definitions have indeed been evaluated. As long as all the needed definitions have been evaluated there is no problem – the update is made, and nothing more needs to be done. A problem arises, however, if not all of the definitions have been evaluated because a deadlock can occur if some special mechanism is not incorporated. This is described below.

As was already mentioned, the only way Poligon allows the user to express serialization is by the serializing of the action parts of rules. The following code fragment executes the changes requested serially;

```
Action Part :
  Changes :
    Change Type : Update
      Updated Node : «aircraft 1»
      Updated Fields :
        wheels ← «a new wheel»
        wings ← «left canard», «right canard»

    Change Type : Update
      Updated Node : «aircraft 2»
      Updated Fields :
        wheels ← «tail wheel»
        wings ← «new left wing»
```

Here the system must wait until the update to «aircraft 1» has finished to execute the second change. In turn, the context executing the rule must wait for confirmation that the first update has occurred from «aircraft 1» before it can perform the update to «aircraft 2». The context, therefore, waits on the stream from which it expects to get the reply confirmation. This is not a problem unless the pseudo-context that was passed on to perform the original update has not already evaluated the required definitions. If such is the case, the execution of the update will require a fully fledged context in order to execute the action part because the code for the evaluation of the outstanding definitions is, by specification, allowed to block for futures at any point.

Since the design of Poligon requires that code executed on blackboard nodes not be allowed to block, the execution of the update has to punt to another context. It is not possible to ask the original context to evaluate the missing definitions because that context is already blocked, waiting for the reply from the update that is in trouble. This is a deadlock condition.[1] A new context is created to perform the update instead, a relatively expensive process; and although the semantics of a rule that punts in this way are very similar to one that does not punt, the system issues a warning message that this is the case so that the programmer can rewrite the rule to force the evaluation of the missing definition. This process is shown in Figure 4-16.

[1]Other mechanisms for deadlock avoidance are possible here, this was just the simplest mechanism given the strongly futures-based implementation model of Poligon.

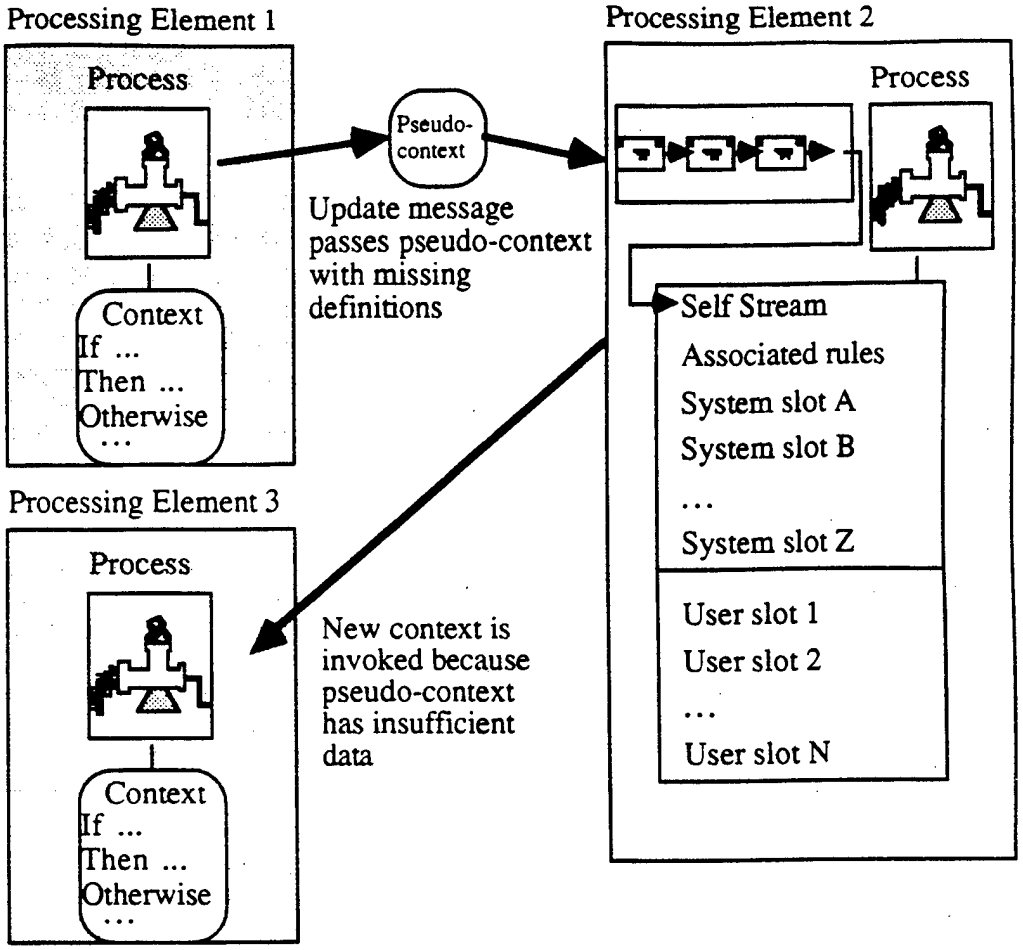


Fig. 4-16. A rule executing in the context on processing element 1 requests an update of a node on processing element 2. The pseudo-context that is passed to the node to be updated does not have enough of its definitions evaluated, so it allocates a new context on processing element 3 to evaluate the missing definitions and to finish the update.

Once the update has been performed either on the node using the pseudo-context or through the agency of a new punted-to context, the original context that is evaluating the rule is sent a message to confirm that this has happened. The context can then synchronize correctly and continue with the evaluation of the serial parts of its rule.[1]

4.7.4. Expectations

The farther the experiment is from theory the closer it is to the Nobel Prize.

— Frédéric Joliot-Curie

Mention was made in Section 4.7.1 that the slots to which rules are attached are defined at compile-time. The exception to prove this rule is discussed in this section.

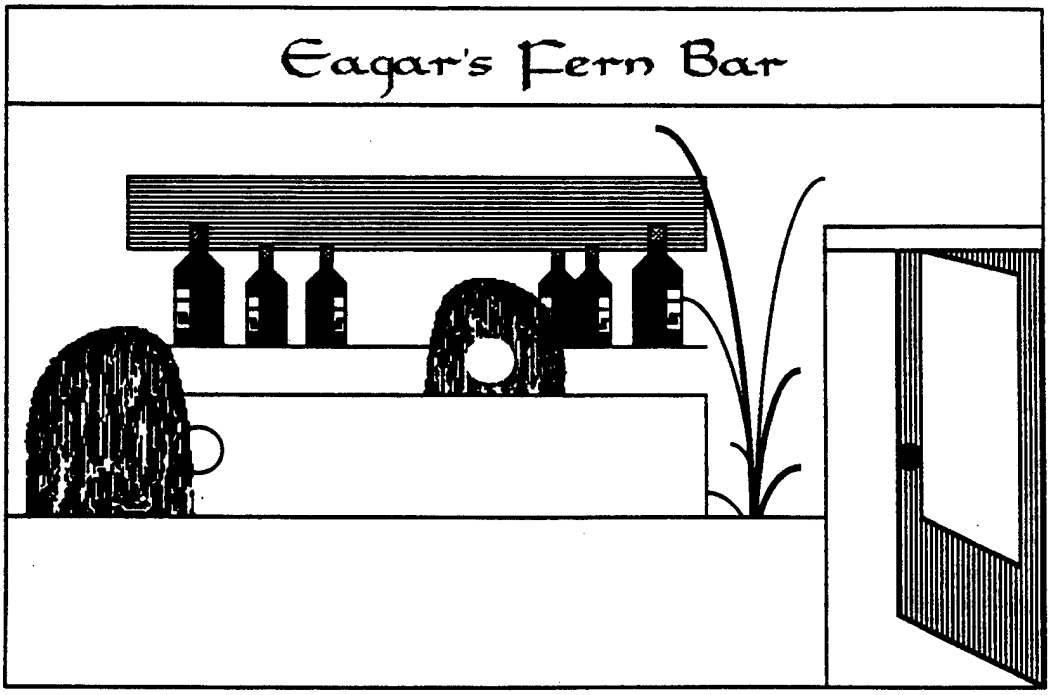
---

[1] In retrospect, this design is inadequate, but deciding on a good implementation for these semantics is difficult. A change to the specified semantics of the language is likely to be the best way around these implementation problems.

An expectation mechanism, a common part of blackboard systems, allows the user to express the fact that some particular event is anticipated. This allows model-based reasoning to focus the attention of the program on places of importance. We wanted to have some way to take advantage of such model based reasoning in Poligon. This section describes the implementation of Poligon's expectations.

Poligon's expectations are dynamically allocated rules. Just as normal Poligon rules are associated with slots at compile-time, expectations are associated with a particular slot on a particular node at run-time. Expectations allow the effective focusing of attention on a particular node while still saving the overhead of associating that rule with all nodes of the class involved. Moreover, state in the rule that initiated the expectation may be used to specialize the rule being allocated so that the behavior can be more highly focused. Because a real-time blackboard system is just as likely to be interested in something not happening as something actually happening, a timeout mechanism supported by Poligon's expectations allows them to wake up after a predetermined time, knowing that they have timed out.

Expectations are represented as structure objects. When a rule decides to post an expectation, it sends a message to the node that will be focused on asking it to record the new expectation. When the expectation is formed, the programmer has the option of defining arguments to the expectation rule and of passing in extra conditions for the When and If parts of the rules. These allow context-specific state to be allocated with the expectation. For instance, if an aircraft is expected to land at a specific airfield, one might post an expectation on the airfield that asked the airfield to look out for aircraft landing. One would also pass it the specific aircraft as an argument to compare with arrivals so that it can know it has noticed the aircraft that is interesting to the program.



Expectation.

Extra slots in the expectation data structure allow the user to specify whether the expectation is active or not or the number of times that the expectation is to try to fire. This is im-

portant because it may be that an expectation is valid for only a highly focused set of circumstances. For instance, the aircraft in question can only land after it has had enough time to fly to the airfield. We do not want expectations to fire off all of the time, and, if the particular aircraft does land, we want the expectation to decouple itself from the node it is watching. All of this is possible in Poligon's expectation mechanism.

Similarly, we may want to know if an event has occurred after a certain time, if the aircraft fails to land at the specified airfield after a given period, for example. This would indicate that our model of the aircraft's actions was wrong and we have to reevaluate what is happening. Such an occurrence is also expressed when the user posts the expectation. The timeout that is to be made is encapsulated on the node being watched, and the set of timeouts that are still pending is examined each time the domain real-time clock ticks. It is an attribute of all Poligon nodes that they can know the real time and that they can be sensitive to clock ticks. When the clock ticks past the time specified by one of the pending timeouts, the rule associated with the expectation is fired and its Timeout Part — a component much like an Action or Otherwise Part — is executed, allowing the program to take whatever corrective action is required.

This timeout mechanism does have a problem, however. In order to work reasonably well, either the system must be lightly loaded or the timeout duration must be long with respect to the time scale of events in the system. This is because a Poligon program can cause the processors on which it runs to become highly and unpredictably loaded, resulting in significant delays in computation. Thus, a timeout might trigger simply because the program was being held up, not because an event failed to occur, i.e., the plane failed to land. Clearly, a Poligon program must not be brittle with respect to timeouts triggering in this way.

At this time, Poligon's expectation mechanism has never been used in an application. This could cause one to lose faith in its usefulness, but we still have some hope for the value of this mechanism, since they may indeed be more useful in a genuinely real-time environment. The knowledge that we implemented for our applications was already expressed in a strongly non-model-based manner, and the real-time aspects of the applications were not really concerned with producing responses that the real-time clock might have triggered. For this reason, the timeout mechanism was not useful. Similarly, in order to be able to compare our experimental results with those produced by the Cage [Aiello 88] and Lamina [Delagi 88b] projects, we were compelled to make the application's problem solving behavior much like that of the others. Because of the way in which the use of expectations affected the problem-solving process, we could not easily produce results comparable to those of the other implementations, so use of the expectation mechanism was removed.

4.8. User Code and Definitions

As you will recall, Poligon supports a mechanism that allows the programmer to express ideas much like knowledge source bindings in an AGE program. These are called *definitions*.

The idea behind definitions was to support the functionality of AGE's knowledge source bindings without suffering from their defects. Nevertheless, Poligon's implementation had a set of defects of its own.

We decided early on that definitions should be lazily evaluated. This was due to an æsthetic preference and a desire to around a deficiency in AGE's implementation. In AGE it is common to define a knowledge source binding with a null value at the knowledge source

level and then to `setq` in a value for it if the value is needed. This substantially complicates the program and obscures the programmer's intent. The plan for Poligon was to have a means of associating names with values and having the expressions that delivered those values execute once at most, but not at all unless the value was needed.

We implemented definitions by making a mapping from names to structures in the context objects that represent the invocation of rules. Thus, for each defined name there is an entry in an AList that associates with that name a value or a token denoting that the definition has not, as yet, been computed and a function that will compute the value if it is needed. In a production-quality Poligon system one could optimize this implementation significantly, but for our purposes an AList was adequate.

When a user made a reference to a definition, the compiler converted it into an access function on the context object that would compute or unpack the value as appropriate. For example,

```
Definitions : four ≡ 2 + 2
When : four = 4
```

would expand during compilation into something of the form

```
When : (eql (get-value-from-definition :four
                _the_current_context_)
            4)
```

where `_the_current_context_` is the name given to the context object that is visible during the evaluation of the When part of the rule. The function `get-value-from-definition` would extract the required value or would compute the value with a function that the compiler generated to represent the expression `2 + 2`.

We quickly found that we had to represent multiple values in definitions, so we used a slightly modified implementation to unpack multiple-valued expressions into their component values.

This implementation was somewhat naïve because it again assumed the existence of an efficient, blackboard machine that would implement this sort of behavior effectively. We discovered, however, that the performance of our applications was being limited by the use of these definition items. The cost of extracting an already computed value from a definition was too high. We could have taken two approaches to address this problem. First, we could have worked to optimize the implementation of definitions, or, second, we could have used a better compiler. Because of the convenience associated with the existing implementation of contexts and their definitions, we decided to try the latter approach.

Our problem derived from code fragments such as the following:

```
Definitions :
    two ≡ 2
    all-wheels ≡ the-aircraft⊕wheels
When : the-aircraft•is-airborne and
       all-wheels•length = two + two
```

In this case example the expression `two + two` makes multiple references to the same definition. From the semantics of definitions we know that each reference to the name `two` will always have the same value. We cannot, however, extract all the definitions in the When-part expression and evaluate them all first because of the short-circuit semantics of operators like And or because of conditional expressions. Not taking notice of these would destroy the lazy semantics of definitions. The compiler was made sensitive to these concerns and transformed the previous expression into something like the following:

```
When :
    (and (get-slot-value the-aircraft :is-airborne)
         (multiple-value-bind (_two_ _all-wheels_)
             (get-multiple-values-from-definitions
               _the_current_context_ :two :all-wheels)
           (eql (length _all-wheels_)
                (+ _two_ _two_))))
```

Here, all of the required definition values for any given lexical level – working outward from deep lexical levels until a non-strict operator or condition is reached –get extracted as a block, and the values are seen in local variables introduced by the compiler. This made a substantial improvement in the performance of user code.

The performance of a system such as Poligon is significantly restricted by the copying of the definitions template into the context objects from the rules to be executed. The definition template is represented as a list of lists, so it is implemented as a `copy-tree` in Poligon. To be sure, this is a suboptimal implementation. A better alternative would be a positional representation of definitions. This would allow the template to be represented as an array, and the initial copying of that template could be a simple block transfer operation. A positional implementation of definitions can easily be made because all the definitions are known at compile-time. The extra effort in reimplementing from the inadequate, original design prevented us from trying it in Poligon.

4.9. Search

Search.

Experience with existing serial blackboard systems such as AGE and MXA caused us to believe that real-world blackboard systems are likely to spend a significant amount of time and effort performing searches over the blackboard. The reason is that these systems frequently need to be able to correlate new pieces of data with the existing solution. For instance, if new blips come into the system from radar detectors, being able to associate these with the aircraft that caused them is important. Making that match often involves searching all the aircraft.

A serial search through the blackboard is usually a linear time process at best. Being able to perform the search in parallel should allow this in principle to happen in constant time, ignoring the overheads and problems associated with parallel processing. We decided, therefore, to support this sort of search operation at the Poligon language level and implement it in as efficient a manner as possible.

Search is implemented in two forms in Poligon that could be efficiently implemented, in a real-world system.[1] Searches over blackboards usually consist of matching a value against a slot in a node. Failing this, they consist of matching some arbitrary condition for a given node. In the first case we were able to encapsulate this request for a match in a simple message to all the nodes being searched. The message contained the value to be compared to, the name of the slot to be compared with, and the operator to be used in the comparison. The second, more general implementation required that a closure be formed over the things to be compared and that the closure be evaluated by each of the potentially matching nodes. This is clearly less efficient than the first case but can still be encoded respectably.

Once we had developed a mechanism for deciding which nodes match, we had only to determine which nodes to search and how to process the replies from the search messages. Poligon, because it is implemented on a machine that supports multicast messages, is able to send the message for the search to a large number of nodes efficiently in one multicast message. Poligon allows the user to search over either a collection of nodes or all the

[1]This is entirely unlike the search mechanism in MXA, which constructed tuples of nodes that matched a certain condition. It was the assumption behind MXA that search would be the overriding cost in the execution of the system, and therefore its model of parallelism required that the blackboard should reside in an associative memory so that these sets of tuples could be constructed efficiently. Unfortunately, no concurrent implementation of MXA was ever made, though the initial candidate for it was an associative database machine.

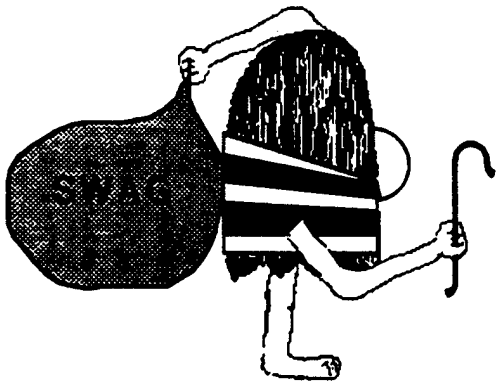
nodes in a class. If the user want to search over a class then the context performing the search sends a message to the class node, which then forwards the search request by multicast to all of the nodes in question.

When the nodes being searched reply to the search request, they all must send messages back to the context that initiated the search. It is not possible for only the matching nodes to reply, because the context performing the search will never know whether it has all of the replies. The context object must therefore have a means of handling all these replies, many of which may simply say "not me." Poligon uses *bags* to implement this behavior. These are described in Section 4.10.1.

4.10. Data Types

Poligon supports some data types not commonly found in other systems: bags, futures and multiple-values objects. In this section we discuss the implementation of these data types, the reasons for their existence, and their benefits, if any.

4.10.1. Bags



Eagar thinks that bags are useful.

Poligon uses bags to handle replies from searches. Bags have the valuable property that they are *unordered* collections of values. This means we can process the values in a bag in any order we want; the ordering does not matter to the program.

When the user performs a search operation such as that specified by the expression

```
Subset of Aircraft Such that Element • wings = 2
```

the value returned immediately is a bag that represents the matching values. This bag is implemented as an object that contains a list of values that have already been determined and a means of generating the values that have not yet been determined. In this case the means of determining the other values is a data type called a *multifuture*. This is a data structure connected to the stream to which all search replies will be sent. As the user tries to extract values from a bag, it returns values from the determined elements first, and if this is empty, it looks on the stream for any new values, only blocking the searcher if there no values are there. If a value found on the stream is a "not me" reply, this is discarded and the bag tries again to get a useful value. The result of this design is that the process looking for the values in the bag will always have access to any new values as soon as they arrive — it doesn't have to wait for a particular element to be returned — and will block only if no values are present.

This implementation gives a clean and abstract interface to the searching process and seems to work very well. Its major deficiencies are that the network can get congested with all the replies to the messages (see Figure 4-17) and that a regular mechanism for implementing collection data types is difficult to optimize without specialized hardware or microcode support.

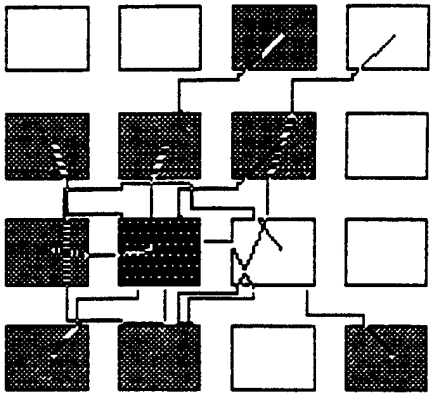


Fig. 4-17. An instrument from the CARE simulator shows the network around a processing element becoming overloaded with a large number of multicast replies.

4.10.2. Multiple Values

Common Lisp supports the return of multiple values from functions. Many implementors of Common Lisp believe that the purpose of multiple values is to avoid CONSing, that is, to save the programmer from having to CONS up a list that denotes the values to be returned and then expecting the caller to unpack the list that was returned.

According to another school of thought, the purpose of multiple values in a language is to express the fact that many functions logically should return more than one value. It is not necessarily meaningful to define a new data structure type for each function's returned values. All the values of a function call are meaningful, however, and there could well be purpose in preserving them. The point here is that in the Common Lisp case, multiple values are CONSed onto the stack and are discarded as soon as possible. In the other case — and Poligon is an example of such a system — multiple values are CONSed into the heap and are discarded as late as possible, i.e., when they encounter a strict operator.

The existence of persistent multiple values is clearly motivated primarily by linguistic æsthetics. Nevertheless, the marginal implementation cost of introducing multiple value objects was small, given that the system already had to be sensitive to strict operators in order to support its model of futures. Multiple values were implemented simply as named structures with a slot containing the list of values. A production-quality system would presumably have a more appropriate implementation. Although somewhat prone to CONSing, this implementation of multiple values seemed to work well, was fairly simple to use, and removed any ambiguity that the transmission of multiple values between processing elements by the system might have caused. There are enough entry points from the user's application into the underlying Poligon implementation that it would have been difficult to preserve the semantics of the native Common Lisp's multiple values implementation when hey were viewed from a Poligon application.

4.10.3. Futures

Unlike existing implementations of systems with futures on real hardware, as opposed to Poligon's simulated machine, Poligon was unable to have low-level support for its implementation of futures. A number of ways in which the Poligon implementation dealt with this issue have already been mentioned. In this section we discuss the implementation of futures themselves.

Poligon supports two forms of future: futures and multifutures. Neither are accessible at the Poligon language level, though when data structures are printed out they often appear in a form somewhat like `(1 2 3 #<Future 4> 5 #<Future Unsatisfied>)`, where `#<Future 4>` is a future that has been satisfied and has the value 4 and `#<Future Unsatisfied>` is a future whose value has not been computed yet or whose value has not yet reached the owner of the future.

Futures in Lamina are not first-class citizens. They are simply specializations of streams that return only one value. They cannot be trivially passed around between objects on different processors. In Poligon, considerable effort was spent on trying to make the implementation of futures as seamless as possible. Futures are named structures that point to the streams that deliver their values. A flag is used to indicate cheaply whether the future has been satisfied or not. Streams live on the particular processing elements on which they were created. As a result, special support has to be provided to allow futures to be passed around between processors.

When an unsatisfied future is passed to another processing element, a remote address to the stream of the originating future is passed along with the future. The copied future therefore has a back pointer to the old stream, but nothing more is done. The future is modified on the originating site so that the future points to a new stream. This stream is then linked to the original stream in the future. It is a property of streams in the CARE machine model that they can be linked, so that values that appear on one stream will be forwarded automatically to any streams linked to it. Thus, if the future that was copied to a different site is ever defutured, it forms a link to the old stream and then waits for any values in the stream to be passed to it. This is a sort of forwarding pointer implementation using message passing across the boundaries of processing elements.

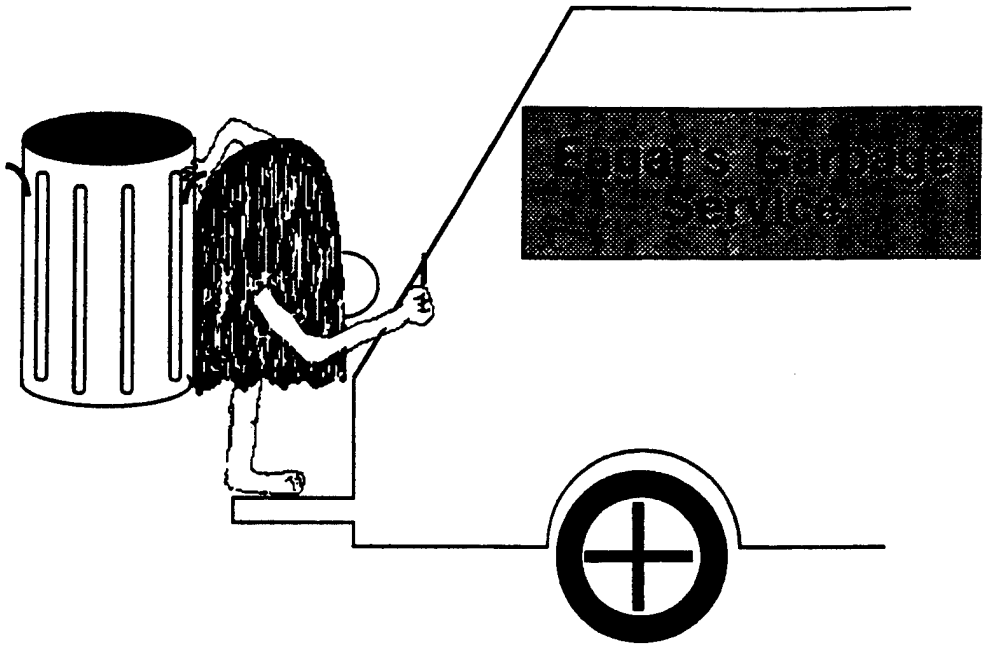
Multifutures implement the bags used in blackboard searching (see Section 4.10.1). They are much like futures in that they are named structures that point to streams. The main difference being that multifutures generally receive more than one message and futures only receive one reply. Multifutures could present a problem to the system in the general case because they do not know how many values they are expecting. They would therefore not know when to relinquish the resources they were using and allow themselves to be garbage collected. In the case of Poligon's use of multifutures, however, the bag that creates the multifuture always knows how many values it is expecting. When the right number of values have returned, the bag can drop the multifuture.

4.11. Optimization

The original implementation, although intended to be highly compilable, was woefully inefficient. This was simply because we were more interested in investigating the concurrent problem-solving process than in making hard measurements of the resulting system's performance. Eventually, however, we had to try to improve the performance of the system in order to get reasonable results from our experiments. These experiments are documented

in some detail in [Nii 88a] and [Rice 88b]. In this section we discuss some of the optimizations we introduced in order to improve Poligon's performance. We developed many of these optimizations to provide efficient support for special cases of generic operations. Consequently, many would not have been necessary if Poligon had not provided as general and abstract a model to the user.

4.11.1. Collections



Collection.

Poligon supports a number of different collection data types, for instance lists, bags, and sets. Because the original implementation of Poligon assumed the existence of specialized hardware to deal with the data types that we wanted to introduce, to implementing a powerful set of generic operations for all collection data types seemed like a good idea. Bags are implemented as Flavors instances, as are sets; lists are just lists. We noticed significant performance degradation in applications from the use of the generic operations that Poligon supports in order to manipulate these data structures. For instance, `foo•head` will extract an element from a bag or the first element from a list. The need to perform numerous `typecases` on everything meant that the application was unable to take advantage of the efficient, microcoded support for list operations. Likewise, because the Poligon system knew little about types, the compiler tended to introduce far more defuturing coercion operators than was strictly necessary. As a result, for almost every argument to every function a Poligon system coercion function was called. This introduced a significant performance penalty.

To alleviate these problems, we implemented considerable support for the declaration, inference, and propagation of types into the compiler. Because the subset of the language in which most of a Poligon program is written is side-effect free, we were able to take advantage of the fact that the type of the value associated with any given identifier does not

change within the scope of that identifier. This means that the compiler was able to propagate inferred and declared types much more effectively.[1]

As an example of this, consider the following expression:

```
Let result ≡ argument•tail In
   Map-Over-A-Collection(#'+, result, 2)
EndLet
```

In the original Poligon implementation this would have compiled into the following Lisp code:

```
(let ((result (tail (↑ argument))))
  (map-over-a-collection #'+ (↑ result) 2))
```

In the above code the function ↑ is the defuturing operator. The `map-over-a-collection` function maps its first argument over the collection denoted by its second argument. As each element is extracted from the collection, it will be put through the ↑ operator in order to apply the + function to it. In the end, a collection that is the same shape as the original collection will be returned. The value of the `result` identifier will be a collection of the same type as that denoted by `argument`, only missing an element.

After the inclusion of the type-checking code we were able to do the following:

```
Has-Type(argument, list, number)◊
Let result ≡ argument•tail
In Map-Over-A-Collection(#'+, result, 2)
EndLet
```

Here the type declaration declares that `argument` is a list of integers. Knowing that the `tail` function is being applied to a list allows the compiler to deduce that `result` must also be a list of numbers. Knowing that `result` is a list of numbers allows the compiler to open code the mapping operation. Similarly, the compiler knows that each element of the list over which it is mapping is a number, so it knows that it will not have to apply any defuturing coercions to the elements of the list during the evaluation of the code. This generates the following lisp code.

```
(let ((result (rest argument)))
  (loop for .element. in result
        collect (+ .element. 2)))
```

Such code will compile into just a few instructions rather than a large and complex set of function calls. The speed improvement in such cases is easily of the order of 20×.

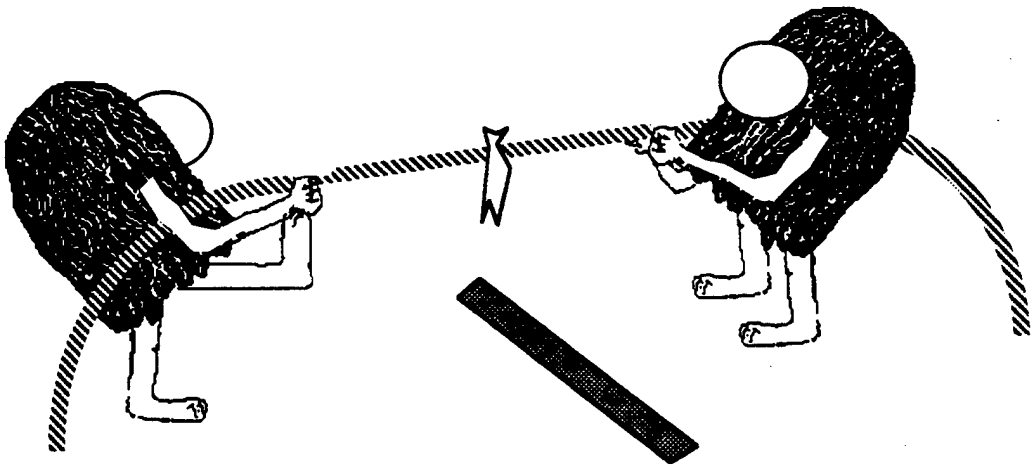
One problem we observed was that the use of such extensive compiler optimization made code harder to debug. Because of the way the compiler open codes Poligon's collection-

[1]It should be noted that this was only possible because we had access to the Lisp compiler's source code, since Common Lisp did not specify any user-accessible interface to the definition of compiler optimizations or to type information. We therefore would like to express our gratitude to Texas Instruments Corp. for its excellent source-code distribution policy.

processing functions, a few lines of source code can often expand into tens of lines of Lisp code, a complication when one lands in the debugger. To address this problem we made the compiler optimizers in Poligon sensitive to the `Optimize` switches in the underlying Lisp system. If the program is compiled for low speed and high safety, then the user gets everything from run-time type checking of named structure accesses and Lisp-level source code debugging. If the code is compiled for high speed and low safety, then run-time type checks are compiled out, structure accesses compile into `aref`s, and collection-processing operations get open coded whenever possible. This use of the Optimize switches proved worthwhile and was used for all Poligon's optimizations.

Clearly, the code shown above is no better than what would have been achieved had the user written everything by hand. Yet, this approach seems useful in practice because the user often does not know which data structures will or will not contain futures or the like, being able to make only occasional assertions about the implementation types of various expressions. This approach seems to decouple the user effectively from the implementation of his data structures and still allows improvements in performance by the addition of type declarations, which do not affect the semantics of the program. This means that optimizations can be achieved without having to rewrite any code.

4.11.2. Equality



Equality.

Poligon needs to have its own idea of equality: it must be able to compare data structures that may have been copied from arbitrary places and get the right answer. Similarly it needs to accommodate special handling for the comparison of futures. It is very undesirable to block on the comparison of futures unless it is strictly necessary. Poligon also needs to have some reasonable behavior to allow the comparison of multiple value objects.

It is for these reasons that the generic Poligon equality-testing operator is very complex and considerably more expensive than the microcoded equality-testing predicates supported on the native machine. The Poligon system was wasting a significant amount of time in making expensive comparisons, so the equality-testing predicate was an early target for compiler optimization. In most cases the use of type declarations and type inference allowed us to compile uses of the = operator into calls to the microcoded `eq` and `eql` operations. Because of this, the examples of generated code in this paper show, references to the = operator as being compiled into calls to the appropriate Lisp predicate.

4.11.3. Slot Reads

Just as type declarations are used to optimize the manipulation of collection data structures, so they are also used to optimize accesses to slots. Poligon originally had a slot read implementation that sent far too many messages. A message was sent to the node asking for the slots to be read; each of these would result in more messages being sent to extract each of the slots involved and then still more messages to get the required values from the slots themselves. This need not be necessary, especially if you know that you are executing code in the process associated with the node, which is always the case for the slot evaluation functions mentioned in Section 4.4.3.

Our approach was to use the type declaration mechanism both to declare the types of the nodes being manipulated, and to declare our knowledge that a given function was to be executed directly by a node. We were, therefore, able to write code such as the following:

```
Define some-function (arg1, node)
Has-Type(arg1, list, number)◊
Has-Type(node, aircraft)◊
  node•wings•length = arg1
EndDefine
```

Now because we know where this code is to be executed, we can compile it into something like the following:[1]

```
(defun some-function (arg1, node)
  (declare (self-flavor aircraft))
  (eql (first (send wings :values)) arg1))
```

Here the message `:values` is being sent to the slot object denoted by the slot name `wings` to extract the values associated with it. Such code would not be necessary in a production-quality system that did not implement its slots as objects being pointed to from its nodes.

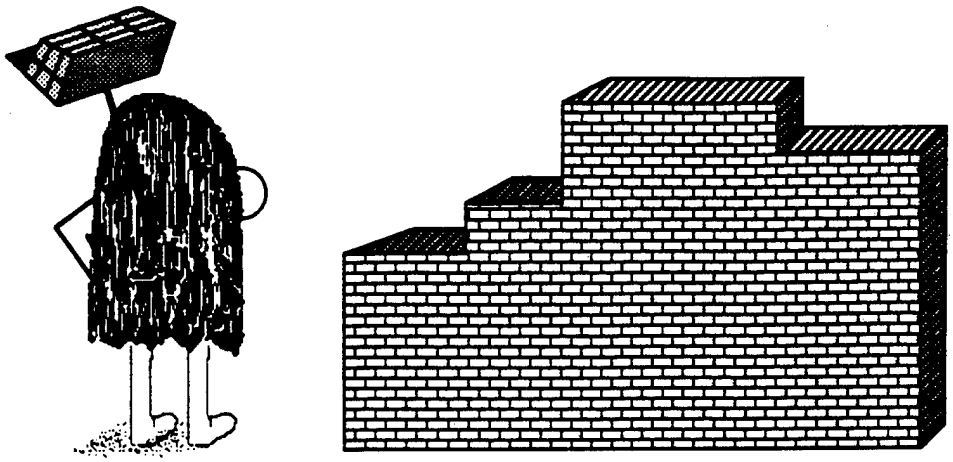
4.11.4. Block Compilation

Block compilation is one aspect of program compilation and optimization that Lisp implementations generally avoid. We did not want to be limited in this way but clearly had to accept that we had no reasonable way of block compiling our Lisp code. We knew, however, that we had the option of block compiling our knowledge base. We found that a significant amount of time was being spent in knowledge search, and this encouraged us to investigate the block compilation of our knowledge base.

In a system such as OPS [Forgy 76], most of the system's time is spent in performing a search over the knowledge base for applicable rules. Blackboard systems, by the very nature of the way in which they are decomposed, have had a substantial amount of knowledge search hand-compiled out. The preconditions on knowledge sources allow the rapid filtering of knowledge and the selection of knowledge sources that are interested in particular classes of events. This same sort of hand compilation applies to Poligon because the user associates rules with particular slots in certain classes of nodes. But, when a slot is

[1]The actual implementation would be more complex than this, but the example shows essentially what happens.

updated in Poligon, the system must still search the (admittedly small) list of associated rules to see whether any rules are interested in the update. This search is not computationally trivial but can be performed at compile-time. Thus, if block compilation is allowed, the search is not necessary.



Block Compilation.

Rules are not associated with the majority of slots in a normal Poligon program. Likewise, most slots that do in fact have associated rules have only a small number, and this number does not change during the execution of the program.[1] Thus, in most cases, it is possible to determine at compile-time all the rules that are interested in all slots. This requires block compilation, since there are always forward references in real code. After the entire knowledge base has been loaded, it is possible to recompile the system so that all slot updates to slots that have no associated rules are open coded in a very simple manner. Slot updates to slots with associated rules are open coded in a manner that wires them directly to the relevant rule objects, thus totally eliminating knowledge search.

This strategy has its costs, however. Once a Poligon program has been block compiled, it is not possible to add or remove a rule without completely recompiling the application. Clearly this is sort of operation would only be performed once an application was well debugged, but it is a small price to pay for improved performance. Moreover, this strategy conforms to Poligon's philosophy, which is to trade extra compilation time for improved run-time performance.

[1] Expectations are an exception to this rule.

4.12. Signal Data Input

> On trapping a lion in a desert [Petard 38]: The Cauchy, or Functiontheoretical, method. *We consider an analytic lion-valued function* f(z). *Let* ζ *be the cage. Consider the integral*
>
> $$\frac{1}{2\pi i}\int_C \frac{f(z)}{z-\zeta}\,dz$$
>
> *where* C *is the boundary of the desert; its value is* f(ζ), *i.e., a lion in the cage.*[1]

Because Poligon is a system that at least attempts to perform real-time operations, we needed a simple mechanism for introducing data into the system. We of course lacked actual real-time monitoring equipment, but we wanted to get as reasonable an implementation as possible.

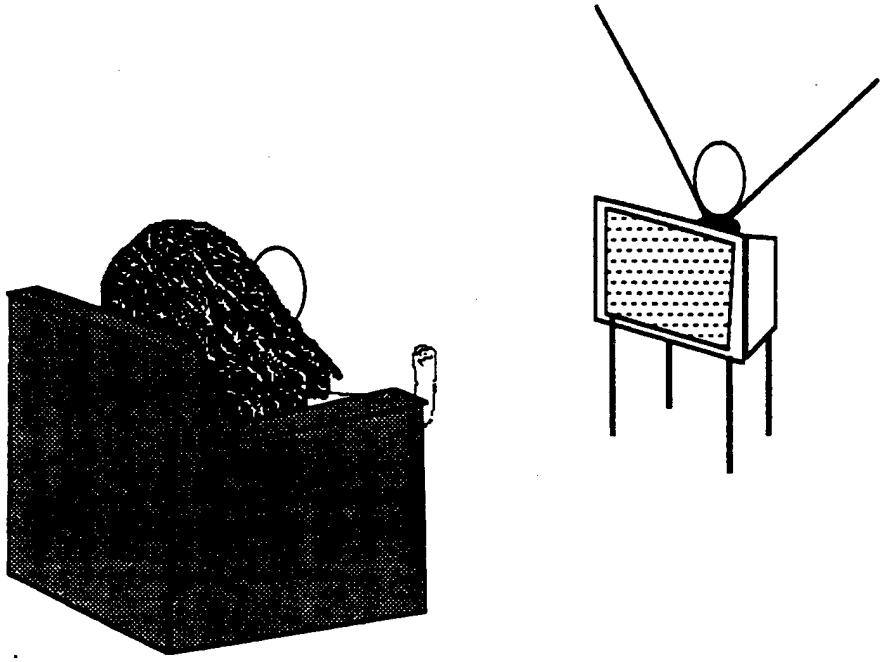
All signal data in Poligon gets into the system from one (Lisp) stream. The data in that stream is timestamped and coded so that the application can find out when the event that it denotes was supposed to have happened and what sort of event it actually represents. This stream is read by the class node of a class of input handlers. The timestamp of the node is read and the node sends a message to itself, which is timed so that the node will wake up to process the signal data at the appropriate simulated domain time. When the class node wakes itself up to deal with the signal data, it allocates to itself an instance of the class that it represents to handle the input from a resource kept in one of its slots. It then sends a message to an instance of itself that tells it to process the input. When an input handler has finished its processing, it sends back a message to the class node on a private stream telling the class node that the input handler is free and can be put back in the resource. If there are not enough instances in the resource to handle the signal data at any time the class node creates new instances and sends them initialization messages that tell them to process the input. In practice, we found that the number of input-handler nodes created was generally the same as the number of signal records read in a given timeslice.

Once the input data arrives at the input-handler server node, a user-defined procedure is invoked in order to process the input. This procedure would typically instantiate a node to represent the input that it had received.

In retrospect, we can see that this implementation had a number of deficiencies. First of all, for linguistic reasons, the only thing these input handlers could really do is create nodes; the user could not update an existing node, for instance. It was effectively always necessary to represent the new data as a node before the system could do anything about it. This is a deficiency because it tends to create a large number of nodes that are not necessarily useful to the representation of the application. In addition, if the programmer wishes to view the process of data arriving in the system as a message-passing process, Poligon will not allow this model.

---

[1] *N.B.* by Picard's Theorem [Osgood 28], we can catch every lion with at most one exception.



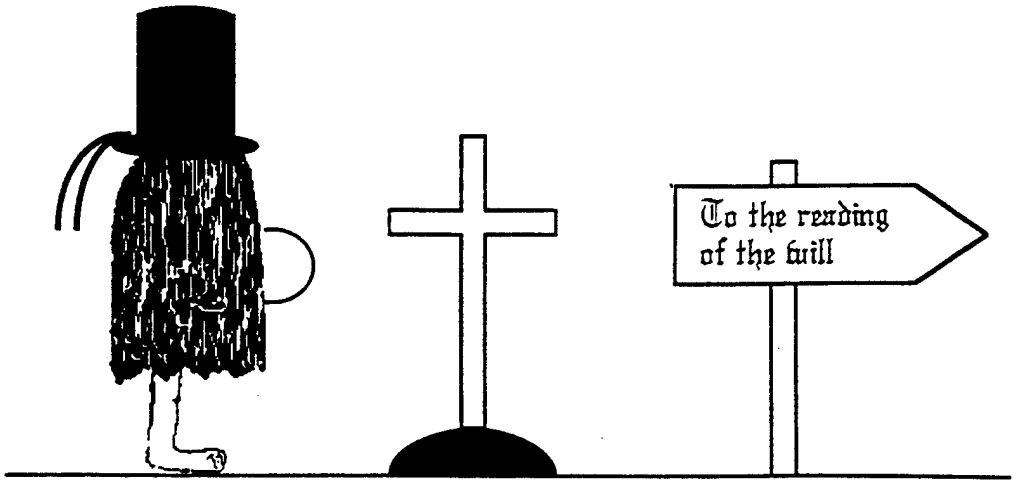
Signal data input.

The second major deficiency of this design is not so much structural as one of execution. It seems worthwhile to create new input handler nodes to deal with new signal data if their resource is empty; but if the system gets heavily congested for whatever reason, the input handler creation process can run away, creating masses of server nodes. In a sense this is an artificial state of affairs, because if the program cannot keep up reasonably well with the real world something is wrong with the program. It is, after all, supposed to be a real-time program. Nevertheless, some limit to the number of input handlers is likely to be needed in order to stop the act of loading data into the system from overwhelming the system. How to compute this limit is a problem we have not addressed. In most of our experiments we were intentionally running the system in a manner that would not overload it excessively. This meant that the system's performance was not adversely affected by input-handling activities. It seems likely, in the absence of any analytic model for Poligon's behavior under such conditions, we would have to perform extensive experimentation in order to find heuristics that would allow us to limit the number of input handlers effectively. This is clearly not a trivial problem since the value would certainly vary with different numbers of processors and varying system load.

4.13. Problem Areas

Early work with Poligon yielded the implementation of considerable functionality, whose ultimate utility was unknown. This is not entirely surprising, given the experimental nature of the project. As Poligon developed, some aspects of the system's behavior gained significant importance; others had to be modified in order to make them useful. Some aspects of the system remained unused and were eventually removed, either because these facilities were never used in our applications and software rot set in or because the initial, naïve implementation was not reasonable in the context of later implementations and our developing understanding of the issues involved. This section concentrates on the aspects of Poligon that did not work as planned.

4.13.1. Property Inheritance and Links



Property inheritance.

A number of existing systems support property inheritance and/or some sort of link mechanism. AGE had a somewhat primitive link mechanism. BB1 supports links along which property inheritance can occur. Many systems also support a limited set of system-defined relationships. For instance, KEE™[1] supports the *instance-of* and the *subclass-of* relationships, as do AGE and BB1, though to a somewhat lesser extent. In developing Poligon, we knew that these relationships would come for free from the implementation. They did not seem sufficient, however, particularly because they do not allow the representation of the *part-of* relationship. Any system like Poligon has a problem with the efficient implementation of relationships. A fixed number of relationships can easily be wired into the system, but any user-defined relationships are likely to be harder to implement and less efficient.

The initial implementation of Poligon came with the *instance-of*, *part-of*, and *subclass-of* relationships built in, and we suspected that we needed something more than this. The initial implementation allowed the inheritance of properties along the *part-of* relationship. Thus, if the program attempted to read a slot on a node that was not present there, the node of which the node in question was a part would be asked for that slot, and so forth up the hierarchy. As this approach appeared insufficient, we proceeded to implement a fairly generalized link mechanism along which property inheritance could also occur. Links were represented as nodes themselves, for reasons of regularity, and system-defined slots on each node would contain a list of these links encapsulated within structures that specified the names of the links. System functions allowed the user to find the nodes linked to a given node by a given relationship.

The links were implemented in such a way that, as property inheritance occurred along a link, a system-defined slot would be triggered so that the user could add rules that were sensitive to the act of property inheritance. This implementation appeared regular, but was very expensive. It also had a number of other deficiencies.

---

[1] KEE is a trademark of Intellicorp.

- The implementation of user-defined links was not the same as that of system-defined relationships, so the same mechanisms could not be used to manipulate these different links in a reasonable way.

- Deciding on the semantics of property inheritance, although the algorithm for it was well defined, caused considerable problems. Because we did not know what behavior was appropriate for this sort of inheritance, the implemented behavior was probably not sensible.

- The inheritance algorithm specified that inheritance would be sought first up the *part-of* links and, failing that, along any user-defined links. The *part-of* relationship in Poligon is under user control, though reasonable default actions occur in setting up this relationship. This means that a node can be directly *part-of* more than one node. It also means that the set of nodes to be searched for inheritance can be circular and that a considerable amount of effort must be expended to avoid being caught in circularities.

- The implementation of inheritance is incompatible with slot access optimization. Unless the user is constrained to state the class of object to be inherited from, slot accesses cannot be optimized.

- With Poligon's current model of stack-group use, the behavior of this inheritance scheme is not implementable in the general case. This is because if a node is asked for a slot that it does not have, it will have to block and ask for the value from elsewhere. Since blocking is not allowed by Poligon nodes, this design cannot be used.[1]

In practice, we found that the property inheritance features were not used even when they worked. This was probably due to a number of factors: the applications appeared not to need it, the abstractions for property inheritance were probably not right, and the inappropriate semantics probably caused users to lose confidence in getting the desired behavior.

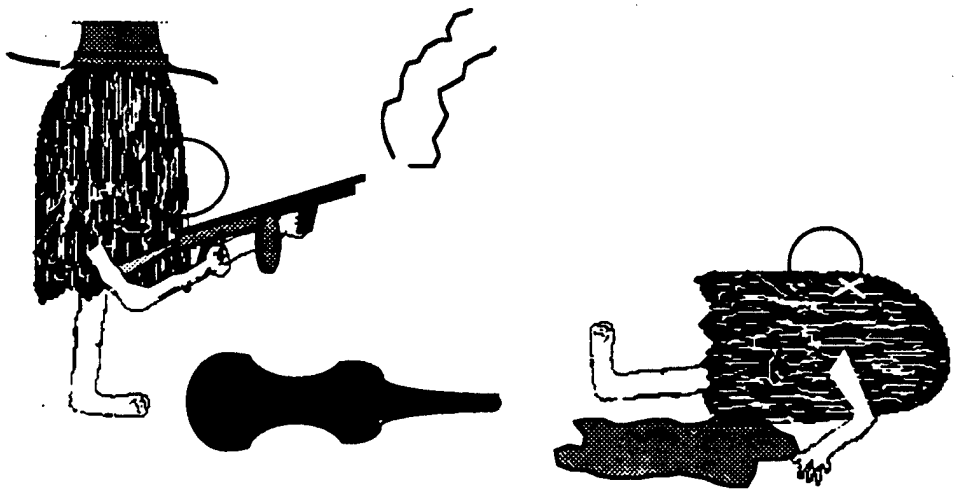
4.13.2. Deletion

The deletion of blackboard nodes seems to be a problem that no blackboard system has really tackled. There is an incompatibility between wanting to have a system that knows all about itself and garbage collection. As a result, the deletion of solution-space elements is usually left to the user. Unfortunately, user-defined deletion and resourcing in a concurrent system is an extremely difficult problem. Conventional models for the deletion of objects rely implicitly on being able to determine that either there are no outstanding references to a node or, more commonly, although there are still outstanding references to a node, none of them will ever be used again. In principle such a state of affairs allows the programmer to recycle nodes, but in Poligon this is not necessarily the case. Although the user may think that no more references are going to be made to a node, it is not possible to determine whether there are any nodes still outstanding for that node backed up somewhere in the

[1] One can envisage a different design in which the node that does not have the slot returns a future to the value of the slot that it doesn't have and sends a message to the node to inherit from, telling that node to reply to the forwarding stream of the future. This works to some extent but prevents the defined semantics of Poligon's slot operations from operating. A multiple slot read or write, for instance, is defined to be atomic. We have no satisfactory way of synchronizing the two nodes in order to get reasonable behavior so the implementation fails.

network. The node might still receive messages that in some way assume it is still the same node even after it has been recycled.

Poligon implemented two forms of deletion for the user: *discarding* and *recycling*. Discarding switched off the node so that no more rules would fire on it but left it able to process any outstanding slot read or write messages. Recycling would completely reinitialize the node and add it to a free list of nodes on the class. The self-stream of the recycled node was replaced with a new one, and the old self-stream was redirected to a site manager object, so that any outstanding messages sent to such a node would be handled in some way — usually by signaling an error.



Deletion.

Although this implementation worked and is still present in Poligon, it was not, in fact, particularly useful for a number of reasons:

- In order for a node to be recyclable, it was necessary to be sure that the reference count to the node was zero. Generally this was possible only if there was just one rule that could fire on the node and if the node would fire that node only once. For this to occur, the node generally had to be at the bottom of the blackboard, created as a result of signal data input.

- As was discussed in Section 4.12, it is not obvious that nodes should be created to represent signal data in the first place. But if they are, the optimized, unmanaged form of node creation should generally be used. Otherwise, large amounts of signal data typically cause the class nodes for the classes created by the signal input procedure to become very hot while servicing all the creation messages. Because the creation of these nodes is usually not managed, it does not have access to the free list of nodes and cannot take advantage of the recycled nodes.

- Discarding nodes generally didn't seem to be useful. It is possible to envisage an application in which the ability to switch nodes off would be useful, but in our applications this did not prove to be the case.

On balance, resource management of blackboard nodes in Poligon is not handled in a useful manner. This is an extremely difficult problem, and possibly the only way to solve it would be to rely on the underlying system's garbage collector.[1]

4.13.3. Messages and Events

Messages were eliminated explicitly from the Poligon language. We did not see any particular use for them since they were an artifact of the implementation, and not part of the blackboard metaphor we were trying to represent. This may well have been an error.

Poligon supports a type of action part in rules called *cause events*. The *cause events* mechanism triggers any rules associated with a slot without actually changing any values in that slot. This mechanism was implemented so that the user could trigger rules without having to perform fake updates to slots, which might have caused errors.[2]

In practice, the *cause events* mechanism was used as a sort of semaphoring idiom and slot updates were often thought of as messages. Consequently, the programmer had to use Poligon's rule mechanism in order to fake messages and methods. It is possible that the programmers were not thinking in a manner appropriate to the blackboard metaphor, but it is equally likely that the Poligon language lacked generality and flexibility in this regard. If we were to try this again we would certainly attempt to find a better abstract model for the integration of message passing, rule invocation and process management.

4.13.4. Load Balancing

On the Advanced Architectures Project, load balancing was originally intended to be managed by a layer of software implemented at a lower level of abstraction than the problem-solving system level. Because of the scale of the project, this issue was not tackled until recently and the work on load balancing did not reach any state of maturity until after all of the experiments with Poligon were over. Thus, all the work on Poligon was based on the assumption that a layer of system software that did not exist would exist at some future time.

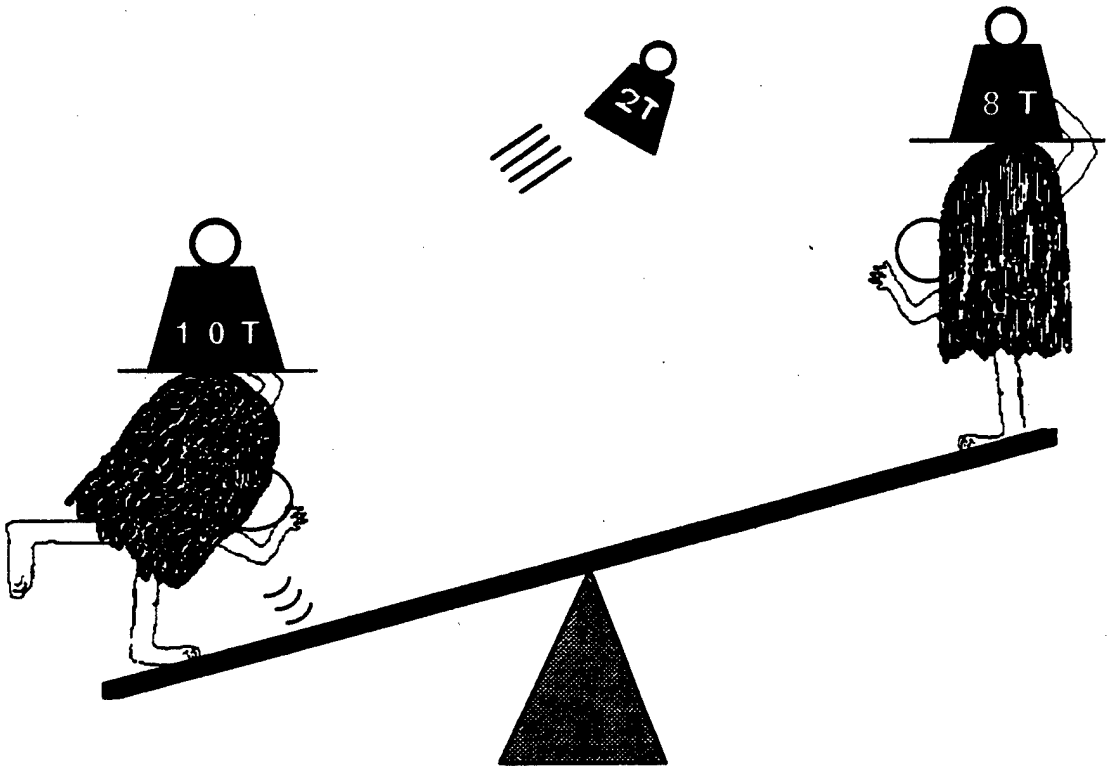
Our experiments showed that load balance is not a trivial issue. We had originally assumed that we could buy back any performance loss from poor load balance by using more processors and thereby lose only efficiency. This proved not to be the case as was shown admirably in the Lamina Elint experiments [Delagi 88b] and also by the Parable experiments [Bandini 89]. The loss in performance from load imbalance proved not only to be substantial but also unrecoverable. Thus, even though we assumed that Silicon would be cheap at the beginning of the project, we found that this was not enough.

---

[1]Garbage collection, incidentally, is a major area that the Advanced Architectures Project has not investigated. We know it to be a difficult problem. Poligon's use of the CARE model improves matters for garbage collection in some ways, since the only objects that are ever transmitted across a processor boundary are remote-addresses or copied data structures with no pointers back to the originating address space. Thus, a garbage collector can always collect any data types other than remote addresses locally. The garbage collection of remote-addresses, however, still remains a major problem. By means of a reference counting model, it seems possible that one could use the CARE processor's communications processor to maintain reference counts as it transmitted remote addresses, but we have not investigated this.

[2]Poligon also supports a means by which the programmer can explicitly state that the rules associated with a particular slot are *not* to be triggered as the result of a specific update.

By design, Poligon did not provide any means for the user to know on which processors a program might be running. It was reasoned that, because of the large number of processors, the problem of load distribution would be sufficiently complex that the machine should be able to out-perform the user.



Load Balancing.

In practice, we found that the user probably should have been given some control over load distribution. For example, the ability to declare that certain classes were likely to create a lot of busy nodes, or that certain class nodes were likely to be very busy.

Ideally, this would be tackled by the environment in some way. It is not difficult to envisage a system that watches the load behavior of a Poligon program and then learns some useful load-distribution heuristics. User declaration of this type would be a second best. Yet, even this model would probably not be sufficient in a fielded system because the problem-solving behavior of the system is so predominantly data dependent. Different behavior in the system will cause wildly differing load characteristics, and so the system would probably need some dynamic load balancing and/or object migration mechanism.

4.13.5. Closures

Closures are one aspect of Poligon that proved to be unnecessarily expensive. This was due to bugs in both the Symbolics and the TI implementations of Common Lisp closures that occurred in the compiling of complex forms such as Poligon application source files. These problems were sufficiently severe that in order to continue with our work we had little choice but to implement our own form of closures. This was not too hard to do because of the semantics of the Poligon language and the existence of the compiler, but the resulting closures, which were implemented as objects, were far less efficient than a native imple-

mentation would have been. Systems like Poligon create a large number of closures. An efficient and bug-free implementation of these is crucial to efficient programming.

4.13.6. Pipelines

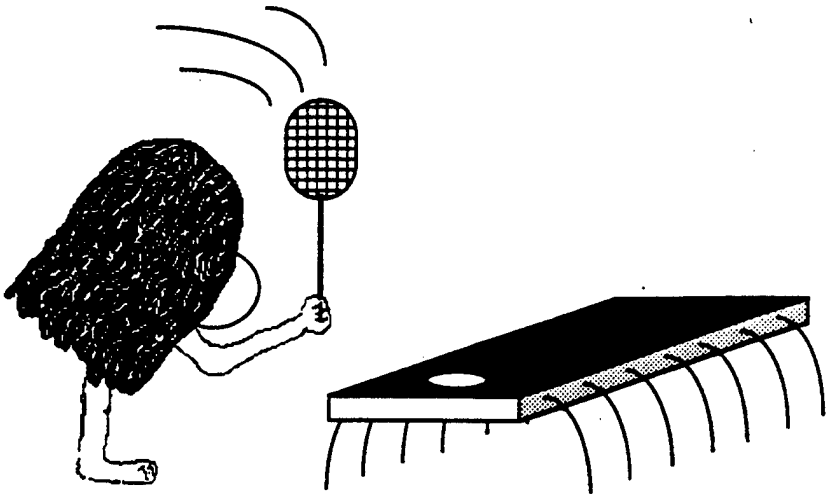
Our early work on Poligon lacked of understanding of the mechanisms by which parallelism is achieved. As a result, we substantially underestimated the importance of pipeline parallelism. In Poligon, pipelines are formed implicitly as data migrates up the abstraction hierarchy, and this may not be the most efficient use of resources. Since the objects in the system communicate with one another by streams, a programming model that encourages the use of non-ephemoral pipelines may be better. This would help to compensate for the cost of stream creation. Lamina [Delagi 86] is just such a programming model. How one could integrate such a programming model with the blackboard metaphor is not at all clear, however. This may be an area in which the underlying system could set up streams between objects and manage them without the user's program having to know about it. We were unable to investigate this area.

4.13.7. Implications for CLOS

The Poligon system was developed using the Flavors object-oriented system before the development of the Common Lisp Object System (CLOS). Although we were generally dedicated to the use of portable standards on the Advanced Architectures Project, we were unable to use them both because they did not exist at the time and also because they would still have been insufficient to give us the level of environmental integration that we sought in Poligon. But, independent of these problems of standardization, there is a more fundamental problem with the new CLOS standard that may not be obvious to the casual reader, but which is likely to be of significance as people start to develop new concurrent problem-solving systems using the evolving standards. CLOS is *unselfish*, that is the concept of *self* has no particular significance in CLOS, unlike Flavors. The behavior of methods is considered to be more closely associated with generic functions than with objects. This has the benefit of giving a regular view of the world and allows multimethods, methods that are specialized on more than one argument.

There is, however, an additional problem with this unifying model; it assumes a shared address space. It is much harder to implement multimethods when the different objects that are being referenced within a method might well be residing in different address spaces on different processors. The implementors of distributed-memory machines tend to think in terms of message passing as the model for communication between both processors and user code. To try to overlay a generic function model of object orientedness on top of this is not a simple matter. We have not had to address this issue because of the immaturity of CLOS, but others in the future will have to think long and hard before they implement a concurrent, object-oriented problem-solving system using CLOS. Certainly, simplifying assumptions can be made. For instance, one could restrict the program only to specialize methods on one argument or only to invoke multimethods on objects that are on the same processing element. Yet each of these simply seems to lead to further complications or loss of generality.

5. Debugging Poligon Programs



Debugging.

Our original motivation in producing Poligon was not just to build a concurrent blackboard system, but rather to build a concurrent blackboard system development tool. Before Poligon was started, a considerable amount of effort had already been expended on the CAOS project [Brown 86] and [Schoen 86]. What we originally assumed would require only a couple of months ended up taking over a year and a half. This was partially due to the immaturity of the CARE simulator, but the difficulty of programming concurrent systems was certainly a major factor. Just as early computer developers would have been hard pressed to envisage a window-based debugging and inspection tool, our first attempts at building concurrent problem-solving systems required investigation in largely unexplored areas. This is especially the case as this work has come before any significant body of expertise has evolved in the debugging of concurrent programs, let alone symbolic programs or problem-solving systems. This section discusses some of the lessons that we learned during the implementation of Poligon and, more importantly, during implementation of applications in Poligon. There are no great pearls of wisdom, but we hope that we can convey which of the features proved useful and which did not.

5.1. Simulation

Our first major observation was that simulation is hard and very time consuming but it is still easier than using real machines. This is due to the flexibility afforded by a simulator,[1] which allows the user to modify the topology, size, and behavior of the machines on which programs are to be run, and also to the inadequacy of the programming environments on existing parallel machines. Having poor development environments is not at all surprising given the comparative youth of these machines, but it was entirely a sufficient reason for not using them in our experiments. The fact that the tools that have been developed for multiprocessors tend to be designed for the debugging of C and FORTRAN programs means that these tools are of little or no use to Lisp programmers.

Because it is much easier to observe the internal behavior of a system in a simulator than on a real machine, we believe that simulation is likely to be an important aspect of program-

[1]The CARE simulator is particularly good in this respect.

ming concurrent systems in the future. A good example of this is what happens when the program dumps you in the debugger.

- On a real parallel machine this presents significant problems. For instance, there is no way of immediately stopping all the processors. Even if the processor that finds the error broadcasts a halt message to the rest of the machine, a considerable amount of extra processing might have happened before the machine comes to rest. This can only confuse things.

- The technology for running debuggers in multiple stack groups on uniprocessors is well developed. This is not necessarily the case with parallel machines. It is much harder to get reasonable behavior out of inspector-like tools that must give a representation of data at random points in the machine. To chase data structures, the inspector will have to make references to remote processors, which could be a problem since it requires a suitable protocol for remote data structure manipulation, even on a distributed memory machine.

- Monitoring message traffic or memory operations on a multiprocessor, although not technologically hard, is hard in practice. This is because it often requires special hardware modification and also because the results delivered from this monitoring is at the wrong level of abstraction for anyone other than the implementors of memory systems or of communications networks. On a uniprocessor running a simulator, monitoring at the appropriate level of abstraction is simple.

- Finally, it should be noted that a crucial aspect of a simulator is that you can modify the simulator itself. Redesigning and then building hardware is a time consuming process. If one can modify a simulator in order to give more debugging information then this, itself, justifies the use of simulation.

5.2. Low-Cost Emulation

An important aspect of Poligon is that it has an emulation mode, Oligon. In this emulation mode, the accuracy and instrumentation of the CARE simulator are given up in favor of an emulation that gives a reasonable facsimile of Poligon's semantics when it is running under CARE, and it does so without a great deal of the cost.

Oligon runs entirely within one stack group. A considerable amount of effort in CARE's simulation is spent in switching stack groups. Oligon does not have to do this because of the way it implements its futures. Oligon's futures look just like Poligon's futures to a Poligon program. Indeed, the user does not even have to recompile a program in order to switch between Oligon and Poligon modes. Internally, however, an Oligon future encapsulates a message that will deliver the value of the future when its method is invoked evaluated.

When an Oligon future is created, it is recorded in a queue of unsatisfied futures. When the user defutures a future by executing a strict operator, the message that will evaluate the future is sent and the future is side-effected with the value of that evaluation.

The serial mode has a simple scheduler that, when it has nothing better to do, executes the messages associated with futures. Thus, all the messages associated with each future are evaluated at some time, which in turn guarantees that the all slot updates and node creation operations will, in fact, happen. This is necessary because futures are used to implement the equivalent Poligon behavior for all interprocess messages in Oligon. The rate at which

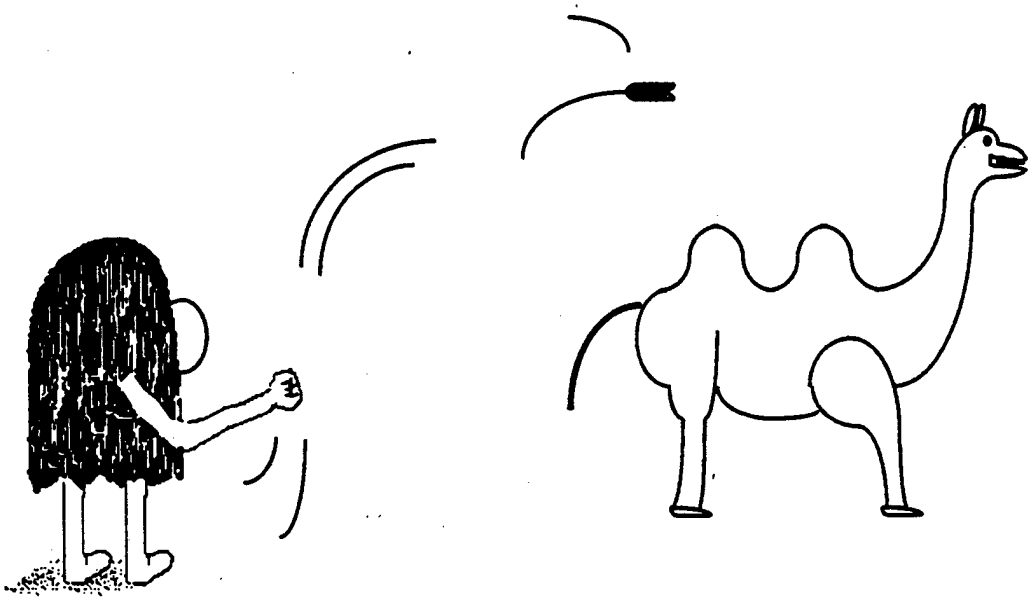
futures are forced by the scheduler is under the user's control, so that the user, to some extent, can emulate different levels of system load.

When a rule attempts to fire, instead of a new process being spun off for the context, a record that points to the context to be invoked is kept in the scheduler queue with all of the arguments that it would have been sent had it been operating in parallel. The scheduler loops around, removing events from this queue. The queue is implemented as a doubly linked list. This allows the user to tell the scheduler to operate in a number of different ways, selecting events to execute in a LIFO, FIFO, or random manner. The use of these different scheduling modes again allows the user to emulate Poligon running in its parallel mode under differing load conditions. The variations in the order in which the rules are in fact executed is sufficiently stressful to the application that once bugs have been eliminated in this emulated mode by means of different scheduler settings, the application will likely run in the Poligon mode without major incident. The sorts of events that are collected in this scheduler queue include the If parts, Action parts and Otherwise parts of rules.

5.3. Trace and Breakpoints

> On trapping a lion in a desert [Petard 38]: The Weiner Tauberian method. *We procure a tame lion,* L_0 *of class* $L(-\infty, \infty)$*, whose Fourier transform nowhere vanishes, and release it in the desert.* L_0 *then converges to our cage. By Wiener's General Tauberian Theorem, [Weiner 33a] any other lion,* L*(say), will then converge to the same cage. Alternatively, we can approximate arbitrarily closely to* L *by translating* L_0 *about the desert [Weiner 33b]*

In the absence of any formal model for the debugging of concurrent blackboard systems, we found it necessary to include debug prints in our code. Although we have not gone much farther than this, we took the step forward that serial systems have made (and perhaps taken this to its logical conclusion) by developing facilities for tracing and breakpoints. It should be noted, however, that although these facilities are optimized for Poligon and the blackboard model, none of them in any way directly address the debugging problems of concurrent systems per se.



Breakpoint.

The native Lisp machines on which Poligon runs provide a number of trace and breakpoint facilities. These are not adequate for our purposes, not because they do not work (they do) but rather because their behavior is at the wrong level of abstraction. The Poligon compiler transforms the user's program into so many different functions and methods that putting trace or breakpoints on these is unlikely to be simple or worthwhile for the user. What the Poligon programmer would prefer is debugging facilities that are closely coupled both to the programming model of the system, and to the common mechanisms by which users introduce errors into their code.

A major problem with tracing activities in a real-time system is that any debugging code is likely to affect the behavior of the program itself. Indeed, it was our experience using MXA that leaving debugging code in is often simpler than trying to debug the real-time behavior of a program once the debugging code has been taken out. To this effect, Poligon tries hard to make its debugging facilities noninvasive. Whenever a trace or breakpoint is entered, the simulated real-time clock is stopped – another benefit of simulation – and started again on exit. Although the cost of executing the code that handles trace and breakpoints is not trivial, the perturbation caused by this code is far smaller than what would have been experienced if it were not possible to stop the clock during the actual processing of the trace. Formatting trace output is so expensive compared to the short evaluations in Poligon, which are typically less than a millisecond, that one cannot afford to count the cost of output of simulations.

Our attempt to provide more focused debugging was a four-pronged attack, first on the knowledge base and then on the blackboard, on general Poligon system activities, and finally on monitoring the program's parallel execution. It should be noted that at any point where a trace point can be applied in Poligon, a breakpoint can generally also be set where this is meaningful.

5.3.1. Debugging Rules

Poligon's rules are split up into a number of different components: the When, If, Select, Action, Otherwise, and Timeout parts. The evaluation of the rules takes place in the context of the set of definitions that have been evaluated up to the relevant point in the execution of the rules. Rules are grouped together in knowledge sources. Even though these are compiled out, in the sense that knowledge sources have no significance in the semantics of the program as it operates, it is still likely that the user will want to view all the rules in a knowledge source together. With a view to these issues we implemented a number of debugging features that are mentioned below and shown in Figure 5-1.

- Any trace or breakpoint operation that can be applied to a rule can also be applied to a knowledge source. This has the effect of applying that operation to all the rules in that knowledge source.

- All of the critical points in a rule are traceable. Thus, trace points can be set on the When, If, Select, Action, Otherwise, and Timeout parts of rules.

- Traces can be set so that the currently evaluated values of definitions are printed out at any point in a rule. This allows the user to monitor the behavior of a rule in terms of the definitions and when they are evaluated.

- Rule failure can be traced. It is a common feature of blackboard systems, and rule-based systems in general, that the user often does not know *why* a given rule fails to fire. The converse is often not the case because it is usually possible to set a

breakpoint in a rule for when it does fire and one can then find out why it fired. Because the user often does not know why a rule failed to fire, we implemented a facility in Poligon allowing the user to set traces on rules that are activated when a particular condition fails to pass. To do this the Poligon compiler takes advantage of the fact that the conditions of rules are usually the conjunction of a number of clauses. The compiler separates out these clauses, and at run time they are executed in the appropriate sequence, checking the trace settings as appropriate. If one of the clauses fails, it can execute the required trace. It is thus possible for the user to set a trace that says *Stop if this rule fails to fire because a clause fails after clause four in the If part.*

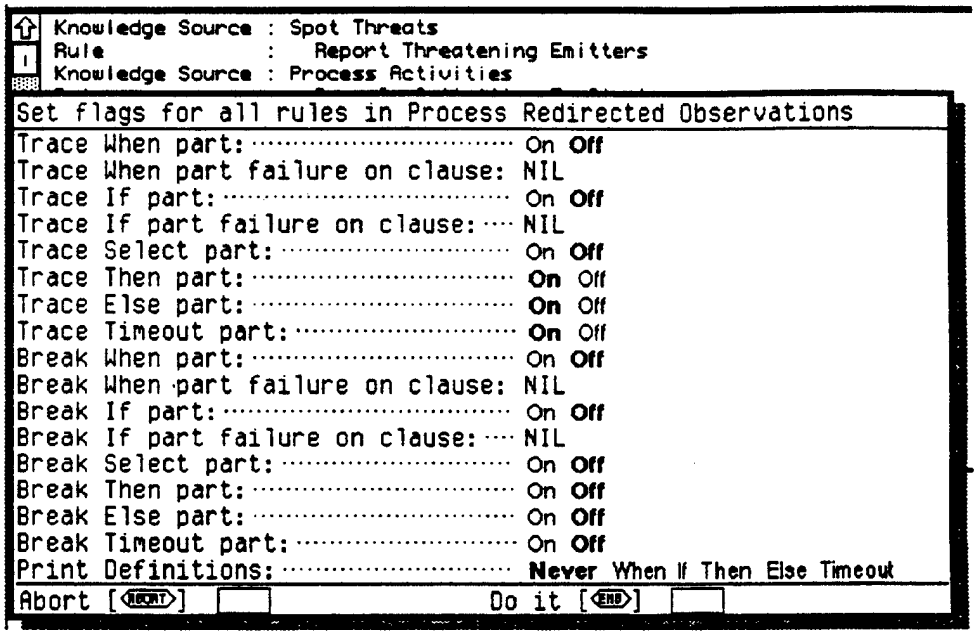


Fig. 5-1. A menu showing the trace and break options available for rules and knowledge sources. In this example a knowledge source has been selected; any trace or breakpoints selected will apply to all rules in that knowledge source.

As is generally the case, these sort of trace features take a certain amount of computation to perform. This is incompatible with the goal of a high-performance system, so these traces are compiled out at high Optimize speed settings, a sacrifice of debugging ease for speed.

5.3.2. Debugging Using Nodes

A number of tracing features have been included within nodes. These in many ways mirror the behavior mentioned earlier for rules and knowledge sources. They are mentioned below and shown in Figures 5-2 and 5-3.

- It is possible to set traces on the reading, writing, or causing of an event on a slot.
- Just as it is possible to set traces for all rules in a knowledge source, it is possible to set a trace that will apply to all instances of a class, or simply to one particular node.

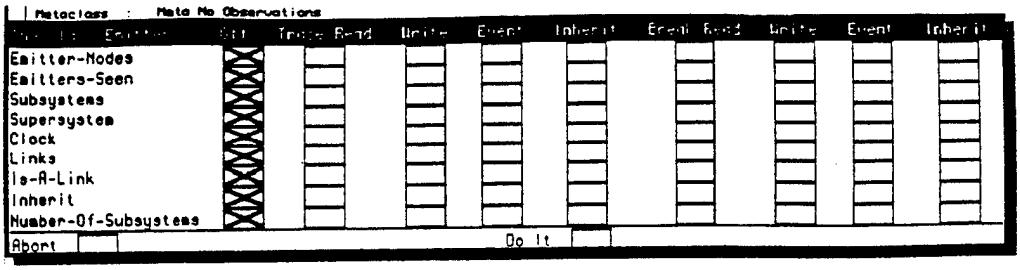


Fig. 5-2. Trace and break options available for operations on Poligon nodes. In this case the class Emitter has been selected. A similar menu allows all instances of a class to have these options set. Through this menu the user can set trace and breakpoints on system-defined slots, such as Number-Of-Subsystems, and on user-defined slots, such as Emitters-Seen.

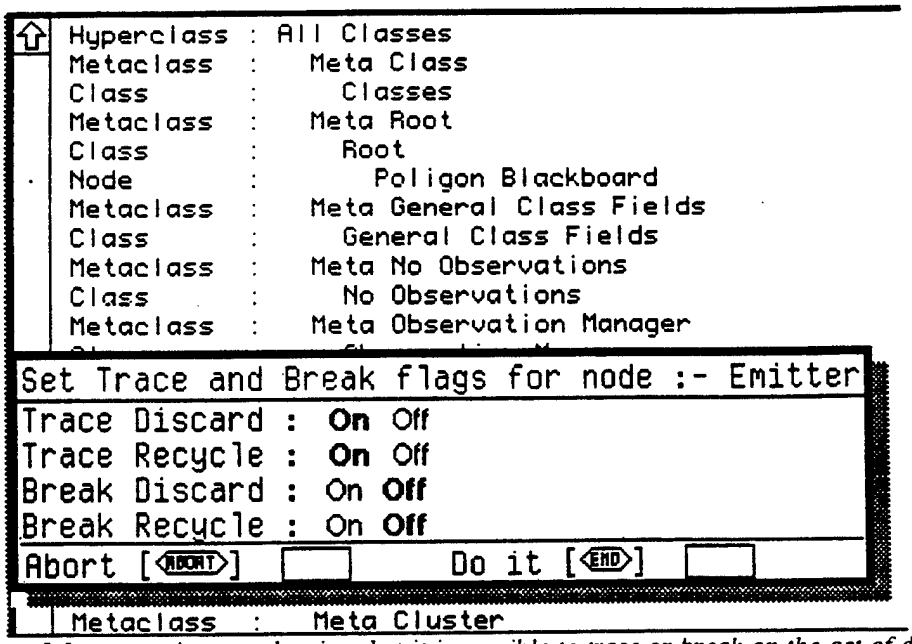


Fig. 5-3. A menu showing that it is possible to trace or break on the act of discarding or recycling a rule.

5.3.3. Tracing System Activities

Technical progress has merely provided us with more efficient means of going backwards.

— Aldous Huxley.

In addition to the trace and breakpoint features just mentioned, a number of trace features allow the user to monitor system functions. It is often the case that the user wants the system to progress to a certain point and then stop. This can be done because Poligon allows breakpoints to be set on the signal records that are read in, on the ticking of the clock, and on the creation of nodes. These are shown in Figure 5-4.

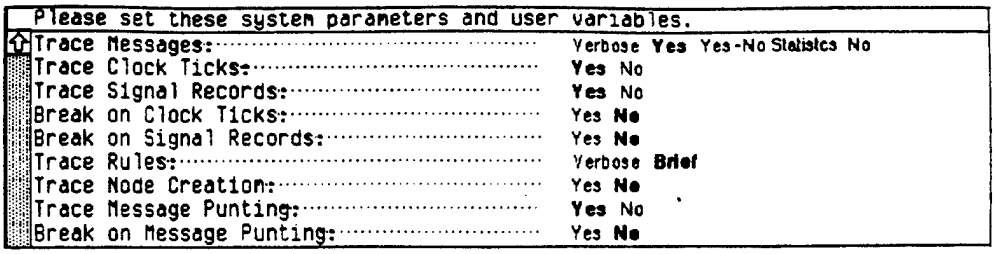


Fig. 5-4. A menu showing that it is possible to trace or break on the act of discarding or recycling a rule.

5.3.4. Monitoring the Parallel Execution of a Poligon Program

A number of additional trace features allow the user to optimize and debug Poligon programs:

- To allow the user to spot and rectify undesirable context punting, traces can be set.
- To allow the user to detect excessively slow pieces of code, the user's code is timed in the Oligon mode and a trace message can be emitted for any code fragment that requires more than a certain time to execute.
- To allow the user to detect slow code in the full, parallel case, Poligon allows the user to trace messages, by recording the messages and the arguments. It times the execution of the messages and allows the user to record only those taking a significant amount of time.
- To allow the user to get detailed information about the behavior of a program, Poligon is interfaced to the native machine's metering package. This allows the detailed metering of user code. Because the metering package can record only a short period of computation, the metering interface allows the user to specify a time to wait before metering commences. This allows the program to progress until it is actually running, as opposed to executing initialization code.

5.4. Perspectives

Finally, we would like to make an observation from our work on Poligon that has general applicability. A Poligon application is instantiated in a large number of data structures that owe their implementation primarily to the search for efficiency, not to intelligibility. It is often the case, therefore, that the programmer's cognitive model of the system may well be entirely different from the implementation model. Consequently, it is crucial to have tools that allow the user to view data structures in a manner that is consistent with the programming model, rather than the implementation model, if rapid debugging is to be possible. There are two simple examples of this in Poligon:

- Contexts have a lot of structure that is used by the system to implement their behavior. For implementation reasons, however, it was difficult to see the values of the definitions that are encapsulated within a context. Fortunately, we had developed an inspector tool that allowed data structures to be viewed simply from different viewpoints. It was therefore simple to define the default behavior for inspecting a context to display it as a mapping from the names of definitions to their values (see

Figures 5-5 and 5-6). Another perspective allows the user to view contexts in terms of their implementation rather than their purpose. Similarly, Poligon nodes are viewed by default in a manner that hides all system details, making it easier for the user to see what the program is really doing (see Figures 5-7 and 5-8). Being able to switch between multiple representations of the same data structure — and, of course, being able to implement these views easily — has proved to be of considerable utility.

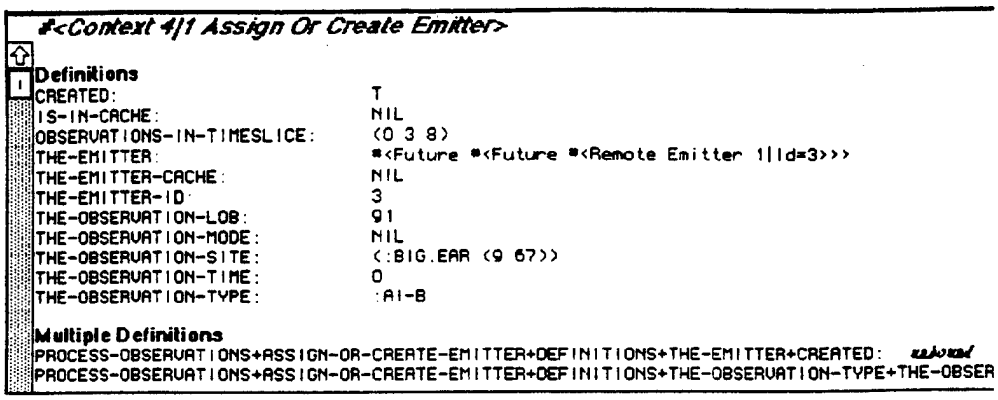


Fig. 5-5. The default perspective for viewing contexts treats them as a means of mapping names into values.

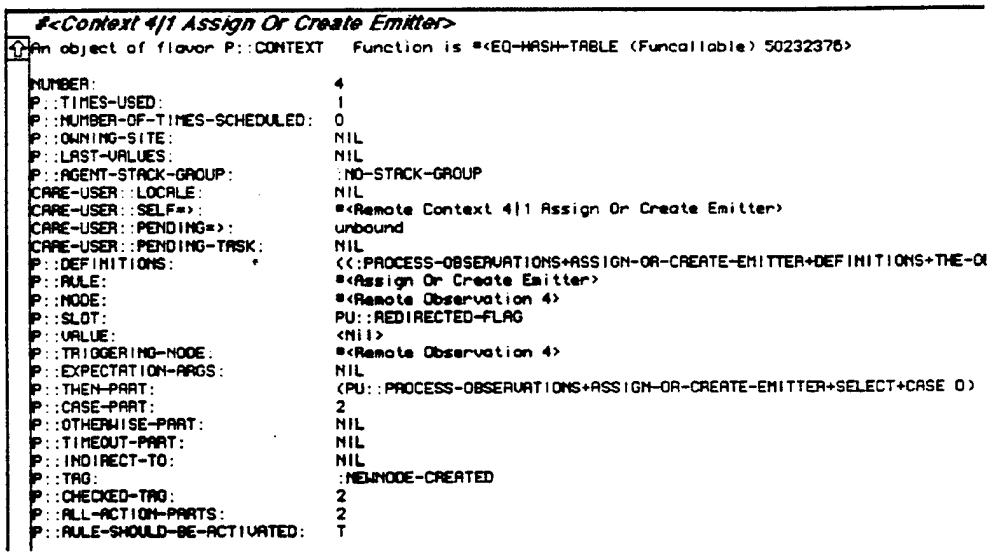


Fig. 5-6. An alternate perspective for viewing contexts allows them to be seen in terms of their implementation.

- In a naïve environment Poligon's implementation makes navigating over the network of Poligon objects very difficult. This is because Poligon's nodes are viewed as remote addresses. These remote addresses point to streams, which in turn point to processes. Somewhere in the context of these processes is some pointer to the actual object that is primarily associated with the process. A large number of mouse clicks in an inspector would therefore be required to get from the remote address of a node to the node that it really points to. Again, fortunately, we had developed tools that allowed us to decouple the printed representation of our data structures from their mouse-sensitive values. Thus, a remote address to a node might be

printed as #<Remote Aircraft-42>. The name Aircraft-42, however, would be mouse-sensitive, and when clicked on, would deliver the node we were interested in. The machine is left to do all the hard work of figuring out what the user wanted to see, a huge saving of effort.

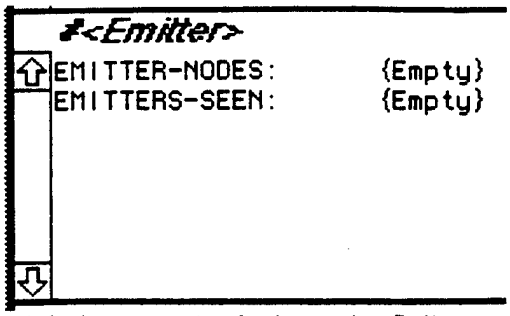


Fig. 5-7. The default perspective for inspecting Poligon nodes causes only user slots to be visible.

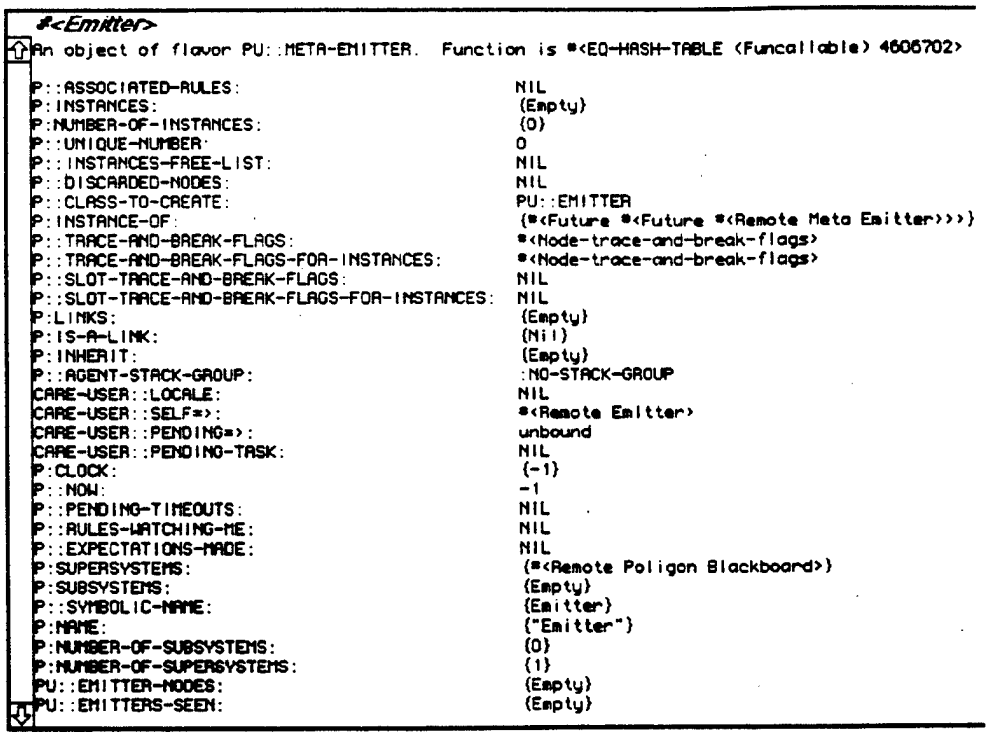


Fig. 5-8. An alternate perspective for viewing Poligon nodes allows the programmer to see the entire system-defined structure of nodes.

5.5. Compiler Optimization

A significant factor in our ability to develop and debug Poligon programs seems to have been the controlled introduction of compiler optimization during debugging. The Poligon programmer has available a number of debugging aids that are progressively switched off as the user asks the compiler for higher levels of optimization. This seems to have been a good decision. The sorts of transformations that are applied to programs, even in conventional, serial systems, can be somewhat counterintuitive and confusing in the process of debugging a program. Clearly, these optimizations should not affect the semantics of a

correct program, so they are only of significance in the presence of program bugs. A number of the optimizations that the Poligon system uses have been discussed above in the related sections. Our intention here is to reiterate our belief in the importance of this design strategy.

6. Conclusions

> *There is altogether no lack in* Genesis *of retribution for failure to obey the Lord. It would not seem, however, that the examples made had much effect. We are thus driven to the conclusion that the direct incentive is more effective than the disincentive, the carrot more useful than the stick. A possible explanation of this fact might be based on the theory that the wrong donkey is beaten every time.*
>
> — C. Northcote Parkinson, *Incentives and Penalties.*

In this paper we have attempted to detail the design and implementation of Poligon, a concurrent problem-solving system modeled closely on the blackboard metaphor.

A number of papers concerning Poligon have focused on its architecture, motivations for its design, its performance, and experiments performed on Poligon applications. None of these publications have indicated how we implemented it or the obstacles encountered and the mistakes we made along the way. This paper described the implementation in sufficient detail that the reader should be able, given enough effort, to implement a system with similar, or better behavior and performance.

We concentrated on Poligon's design as an example of an attempt to develop a high performance, concurrent problem-solving system. We have delineated a set of issues that implementors must address in order to achieve good performance in a concurrent blackboard system. Many of these observations are applicable to other architectures and to serial systems as well. The important aspects are node creation, knowledge search, conflict resolution, knowledge invocation, context evaluation, slot reads, slot updates, event posting, and the efficient handling of stack groups and processes. Each of these aspects of a system's performance were discussed with particular reference to the Poligon model.

A number of features in Poligon proved to be inadequate, difficult, or didn't work at all. Among these were run-time property inheritance, node deletion and reuse, message passing in a rule-based system, the efficient use of pipelines, and load balancing. We also found ourselves unable to shield the user from the differing costs of communicating with local versus remote memory.

We have found that the blackboard model, an appealing cognitive model for concurrent problem solving, does not necessarily work as well in practice as intuition might lead one to expect. As a consequence, although we hoped to deliver many orders of magnitude of speedup due to parallelism, we have only been able to show about one order of magnitude and there are indications that this might scale to about two orders of magnitude. When comparing our application against the same application written in AGE, however, we observe that the application's simulated performance in Poligon was about fifteen thousand times faster. At least by comparison, therefore, we can assert that we have, indeed, built a high-performance concurrent blackboard tool.

In countries with an aristocratic tradition (like Britain) the highest status is associated with official position, birth, education, athletic prowess and gallantry in battle. In countries without any such tradition (like U.S.A.) the highest status is associated with the biggest capital and income. Very seldom do we meet a millionaire with a V.C., and Sir Thomas More's achievement, in being both knighted and canonized, is likely to remain an unbeaten record.

— C. Northcote Parkinson, *Incentives and Penalties.*

7. Bibliography

[Aiello 86] Aiello, Nelleke. *User-Directed Control of Parallelism: The Cage System.* Technical Report KSL–86–31, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986.

[Aiello 88] Aiello, Nelleke. *Cage: The Performance of a Concurrent Blackboard Environment.* Technical Report KSL–88–80, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.

[Bandini 89] Bandini, Jean-Christophe. *Poligon Applications.* Technical Report KSL–89–43, Heuristic Programming Project, Computer Science Department, Stanford University, 1989.

[Brown 86] Brown, Harold, Eric Schoen, and Bruce A. Delagi. *An Experiment in Knowledge-Base Signal Understanding Using Parallel Architectures.* Technical Report STAN–CS–86–1136, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.

[Byrd 88] Byrd, Gregory T. and Bruce A. Delagi. *A Performance Comparison of Shared Variables vs. Message Passing.* Technical Report KSL–88–10, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of Third International Conference on Supercomputing, Vol. 1, pages 1–7, Boston, MA, March 1988 International Supercomputing Institute.

[Delagi 86] Delagi, Bruce A., Nakul P. Saraiya and Gregory T. Byrd. *LAMINA: CARE Applications Interface.* Technical Report KSL–86–76, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

[Delagi 88a] Delagi, Bruce A., Nakul P. Saraiya, Gregory T. Byrd, and Sayuri Nishimura. *CARE User's Manual.* Technical Report KSL–88–53, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.

[Delagi 88b] Bruce A. Delagi and Nakul P. Saraiya. *ELINT in LAMINA: Application of a Concurrent Object Language.* Technical Report KSL–88–33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in SIGPLAN Notices, February 1989.

[Erman 80] Erman, Lee D., Frederick Hayes-Roth, Victor R. Lesser, D. Raj Reddy. *The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty*. ACM Computing Survey. 12: 213–253. June 1980.

[Engelmore 88] Engelmore, Robert and Tony Morgan, eds. *Blackboard Systems*. Addison-Wesley Publishing Company Inc., Menlo Park , CA 1988.

[Forgy 76] Forgy, C. and J. McDermott. *The OPS Reference Manual*. Carnegie-Mellon University. 1976.

[Gupta 86] Gupta Anoop. *Parallelism in Production Systems*. Technical Report, Computer Science Department, Carnegie-Mellon University, March, 1986. Ph. D. dissertation.

[Halstead 84] Halstead, R. H. Jr. *Implementation of Multilisp: Lisp on a Multiprocessor*. Proceedings of the ACM Symposium on Lisp and Functional Programming. 9–17, August 1984.

[Hayes-Roth 85] Hayes-Roth, B. *Blackboard Architecture for Control*. Journal of Artificial Intelligence. 26: 251— 321. 1985.

[Hillis 85] Hillis, W. D. *The Connection Machine*. MIT Press. Cambridge, MA. 1985.

[Nii 79] Nii, H. Penny, and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs*. Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645–655, 1979.

[Nii 82] Nii, H. P., E. A. Feigenbaum, J. J. Anton, and A. J. Rockmore. *Signal-to-Symbol Transformation: HASP/SIAP Case Study*. Technical Report HPP–82–6, Heuristic Programming Project, Computer Science Department, Stanford University, 1982. Also in AI Magazine. 3:2, 23–35, 1982.

[Nii 86] Nii, H. Penny. *Blackboard Systems*. Technical Report KSL–86–18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in AI Magazine, 7:2 and vol. 7:3, 1986.

[Nii 88a] Nii, H. Penny, Nelleke Aiello and James Rice. *Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon*. Technical Report KSL–88–02, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1988. Also in [Engelmore 88].

[Nii 88b] H. Penny Nii, Nelleke Aiello, James Rice. *Experiments on Cage and Poligon: Measuring the performance of Parallel Blackboard Systems*. Technical Report KSL–88–66, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in *Distributed Artificial Intelligence II*. L. Gasser and M. N. Huhns (eds). Morgan Kaufmann, San Mateo, CA 1989.

[Osgood 28] W. F. Osgood. *Lehbuch der Funktionentheorie*, vol 1 (1928) p 178.

[Petard 38] H. Petard, "A Contribution to the Mathematical Theory of Big Game Hunting," *American Mathematical Monthly* 45:446, 1938.

[Rice 84] Rice, James. *The MXA user's and writer's companion.* Systems Programming Ltd., The Charter Abingdon, Oxon, U.K. 1984.

[Rice 86] Rice, James. *The Poligon User's Manual.* Technical Report KSL–86–10, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.

[Rice 88a] Rice, James. *Problems with Problem-Solving in Parallel: The Poligon System.* Technical Report KSL–88–04, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of Third International Conference on Supercomputing, pages 25–34, Boston, MA, March 1988 International Supercomputing Institute, and *Artificial Intelligence, Simulation and Modelling*, Lawrence Widman (ed), John Wiley Publishing Company, New York 1989.

[Rice 88b] Rice, James. *The Elint Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System.* Technical Report KSL–88–69, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of International Joint Conference on Artificial Intelligence, Vol. 1, 212–217, 1989.

[Rice 88c] Rice, James. *The Advanced Architectures Project.* Technical Report KSL–88–71, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.

[Saraiya 89] Saraiya, Nakul P. *Design and Performance Evaluation of a Parallel Report Integration System.* Technical Report KSL–89–16, Heuristic Programming Project, Computer Science Department, Stanford University, April 1989.

[Schoen 86] Schoen, Eric. *The CAOS System.* Technical Report KSL–86–22, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, 160–170, April 1986.

[Weiner 33a] Weiner, Norbert. *The Fourier Integral and Certain of Its Applications* (1933) pp 73–74.

[Weiner 33b] Weiner, Norbert. *The Fourier Integral and Certain of Its Applications* (1933) p 89.

My idea of an agreeable person is a person who agrees with me.

— Benjamin Disraeli

Reproduced by NTIS

National Technical Information Service
Springfield, VA 22161

This report was printed specifically for your order from nearly 3 million titles available in our collection.

For economy and efficiency, NTIS does not maintain stock of its vast collection of technical reports. Rather, most documents are printed for each order. Documents that are not in electronic format are reproduced from master archival copies and are the best possible reproductions available. If you have any questions concerning this document or any order you have placed with NTIS, please call our Customer Service Department at (703) 487-4660.

About NTIS

NTIS collects scientific, technical, engineering, and business related information — then organizes, maintains, and disseminates that information in a variety of formats — from microfiche to online services. The NTIS collection of nearly 3 million titles includes reports describing research conducted or sponsored by federal agencies and their contractors; statistical and business information; U.S. military publications; audiovisual products; computer software and electronic databases developed by federal agencies; training tools; and technical reports prepared by research organizations worldwide. Approximately 100,000 *new* titles are added and indexed into the NTIS collection annually.

For more information about NTIS products and services, call NTIS at (703) 487-4650 and request the free *NTIS Catalog of Products and Services*, PR-827LPG, or visit the NTIS Web site **http://www.ntis.gov**.

NTIS
Your indispensable resource for government-sponsored information—U.S. and worldwide